Computational Physics With Python

Dr. Eric Ayars

California State University, Chico

Version 0.9, August 18, 2013

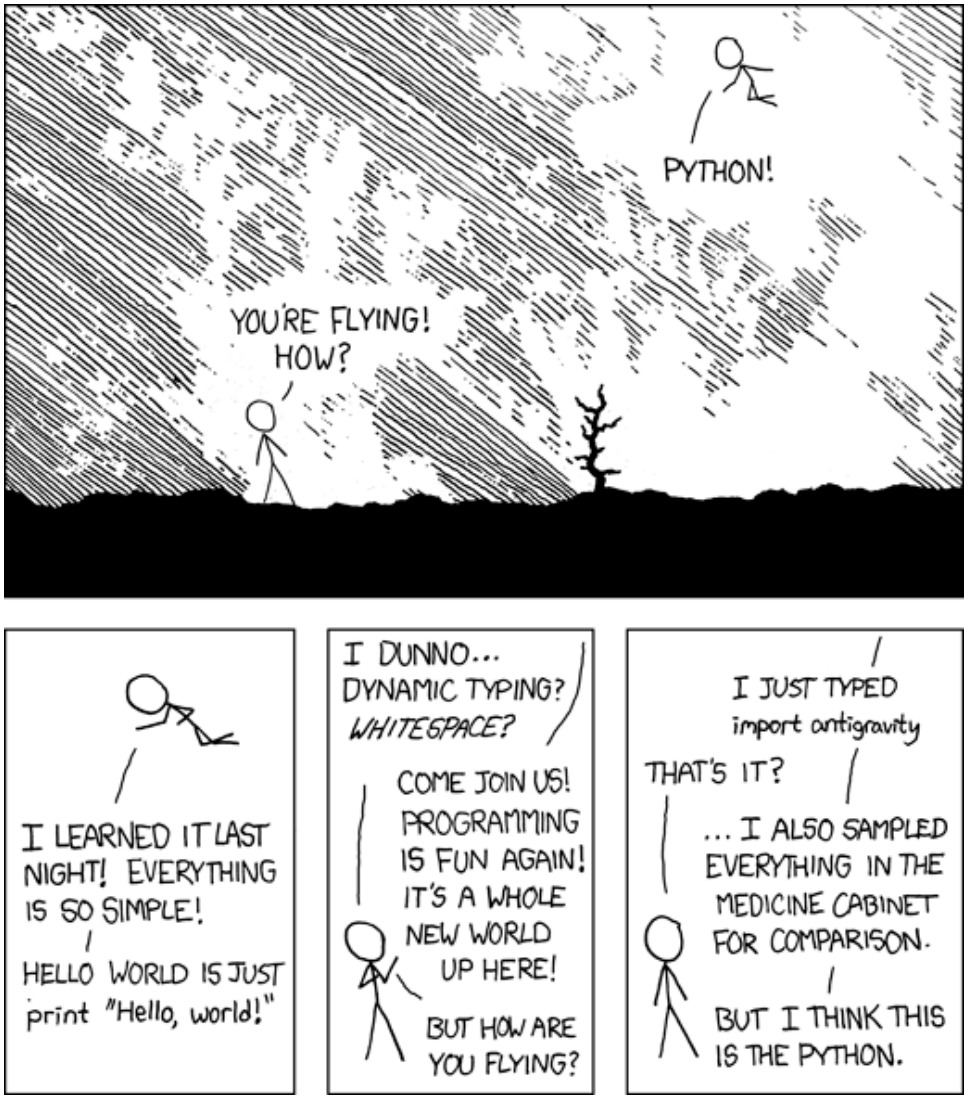
Contents

Preface: Why Python?

When I began teaching computational physics, the first decision facing me was "which language do I use?" With the sheer number of good programming languages available, it was not an obvious choice. I wanted to teach the course with a general-purpose language, so that students could easily take advantage of the skills they gained in the course in fields outside of physics. The language had to be readily available on all major operating systems. Finally, the language had to be *free*. I wanted to provide the students with a skill that they did not have to pay to use!

It was roughly a month before my first computational physics course began that I was introduced to Python by Bruce Sherwood and Ruth Chabay, and I realized immediately that this was the language I needed for my course. It is simple and easy to learn; it's also easy to *read* what another programmer has written in Python and figure out what it does. Its whitespace-specific formatting forces new programmers to write readable code. There are numeric libraries available with just what I needed for the course. It's free and available on all major operating systems. And although it is simple enough to allow students with no prior programming experience to solve interesting problems early in the course, it's powerful enough to be used for "serious" numeric work in physics — and it *is* used for just this by the astrophysics community.

Finally, Python is named for my favorite British comedy troupe. What's not to like?


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Chapter 0

Useful Introductory Python

0.0 Making graphs

Python is a scripting language. A script consists of a list of commands, which the Python interpreter changes into machine code one line at a time. Those lines are then executed by the computer.

For most of this course we'll be putting together long lists of fairly complicated commands —programs— and trying to make those programs do something useful for us. But as an appetizer, let's take a look at using Python with individual commands, rather than entire programs; we can still try to make those commands useful!

Start by opening a terminal window.[1] Start an interactive Python session, with pylab extensions[2], by typing the command ipython −−pylab followed by a return. After a few seconds, you will see a welcome message and a prompt:

```
In [1]:
```

Since this chapter is presumbly about graphing, let's start by giving Python something to graph:

```
In [1]: x = array([1,2,3,4,5])
In [2]: y = x+3
```

[1] In all examples, this book will assume that you are using a Unix-based computer: either Linux or Macintosh. If you are using a Windows machine and are for some reason unable or unwilling to upgrade that machine to Linux, you can still use Python on a command line by installing the Python(x,y) package and opening an "iPython" window.

[2] All this terminology will be explained eventually. For now, just use it and enjoy the results.

Next, we'll tell Python to graph y versus x, using red $\times$ symbols:

```
In [3]: plot(x,y,'rx')
Out[3]: [<matplotlib.lines.Line2D at (gibberish)>]
```

In addition to the nearly useless Out[] statement in your terminal window, you will note that a new window opens showing a graph with red $\times$'s.

The graph is ugly, so let's clean it up a bit. Enter the following commands at the iPython prompt, and see what they do to the graph window: (I've left out the In []: and Out[]: prompts.)

```
title('My first graph')
xlabel('Time (fortnights)')
ylabel('Distance (furlongs)')
xlim(0, 6)
ylim(0, 10)
```

In the end, you should get something that looks like figure 0.

Let's take a moment to talk about what's we've done so far. For starters, x and y are *variables*. Variables in Python are essentially storage bins: x in this case is an address which points to a memory bin somewhere in the computer that contains an *array* of 5 numbers. Python variables can point to bins containing just about anything: different types of numbers, lists, files on the hard drive, strings of text characters, true/false values, other bits of Python code, *whatever*! When any other line in the Python script refers to a variable, Python looks at the appropriate memory bin and pulls out those contents. When Python gets our second line

```
In [2]: y = x+3
```

It pulls out the x array, adds three to everything in that array, puts the resulting array in another memory bin, and makes y point to that new bin.

The plot command plot(x,y,'rx') creates a new figure window if none exists, then makes a graph in that window. The first item in parenthesis is the x data, the second is the y data, and the third is a description of how the data should be represented on the graph, in this case red $\times$ symbols.

Here's a more complex example to try. Entering these commands at the iPython prompt will give you a graph like figure 1:

```
time = linspace(0.0, 10.0, 100)
height = exp(-time/3.0)*sin(time*3)
figure()
```

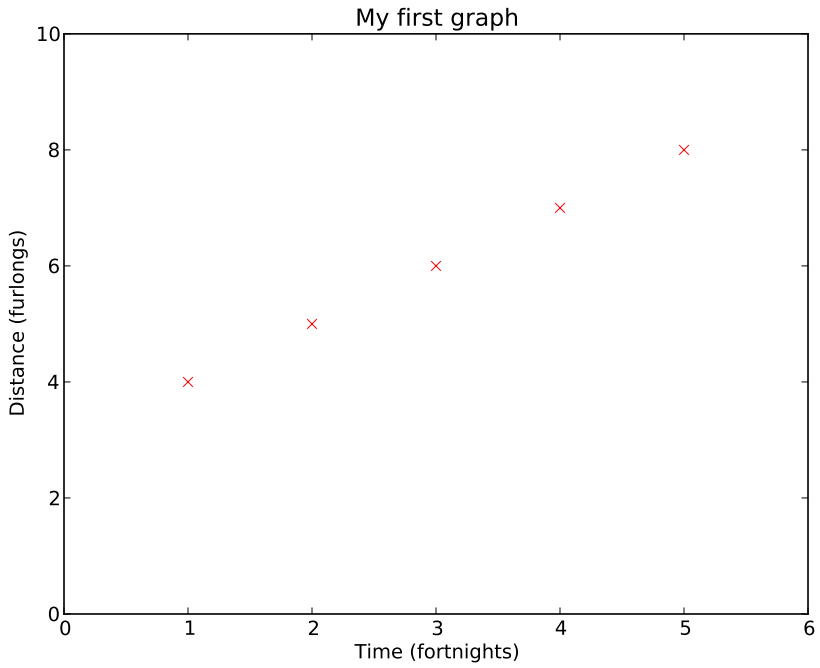


Figure 0: A simple graph made interactively with iPython.

```
plot(time, height, 'm-^')
plot(time, 0.3*sin(time*3), 'g-')
legend(['damped', 'constant amplitude'], loc='upper right')
xlabel('Time (s)')
```

The linspace() function is very useful. Instead of having to type in values for all the time axis points, we just tell Python that we want linearly-spaced numbers from (in this case) 0.0 through 10.0, and we want 100 of them. This makes a nice x-axis for the graph. The second line makes an array called 'height', each element of which is calculated from the corresponding element in 'time' The figure() command makes a new figure window. The first plot command is straightforward (with some new color and symbol indicators), but the second plot line is different. In that second line we just put a calculation in place of our y values. This is perfectly fine with Python: it just needs an array there, and does not care whether it's an array that was retrieved from a memory bin (i.e. 'height') or an array calculated on the

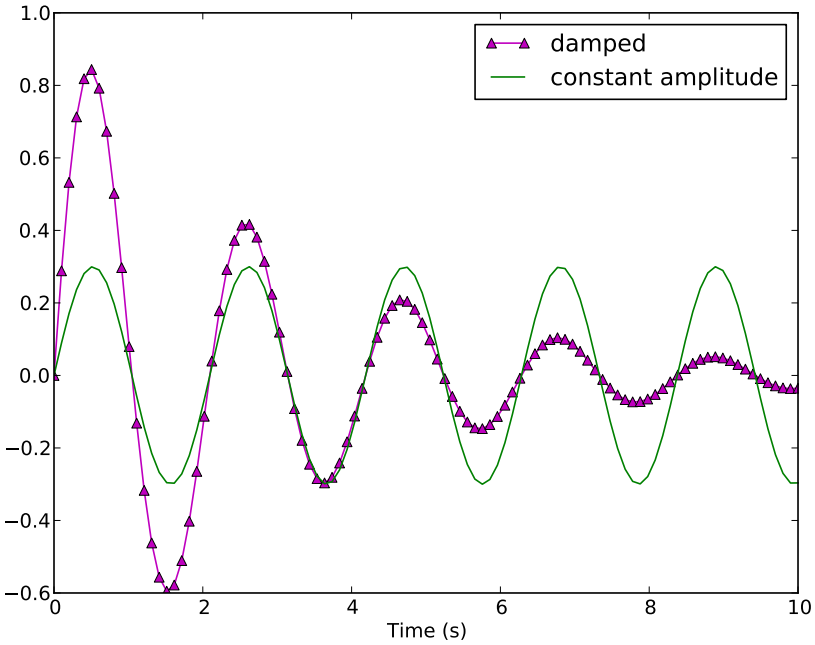


Figure 1: More complicated graphing example.

spot. The legend() command was given two parameters. The first parameter is a *list*[3]:

```
['damped', 'constant amplitude']
```

Lists are indicated with square brackets, and the list elements are separated by commas. In this list, the two list elements are strings; strings are sequences of characters delimited (generally) by either single or double quotes. The second parameter in the legend() call is a labeled option: these are often built in to functions where it's desirable to build the functions with a default value but still have the option of changing that value if needed[4].

[3]See section 1.3.
[4]See section 1.7.

0.1 Libraries

By itself, Python does not do plots. It doesn't even do trig functions or square roots. But when you start iPython with the '-pylab' option, you are telling it to load optional *libraries* that expand the functionality of the Python language. The specific libraries loaded by '-pylab' are mathematical and scientific in nature; but Python libraries are available to read web pages, create 3D animations, parse XML files, pilot autonomous aircraft, and just about anything else you can imagine. It's easy to make libraries in Python, and you'll learn how as you work your way through this class. But you will find that for many problems someone has already written a Python library that solves the problem, and the quickest and best way of solving the problem is to figure out how to use their library!

For plotting, the preferred Python library is "matplotlib". That's the library being used for the plots you've made in this chapter so far; but we've barely scratched the surface of what the matplotlib library is capable of doing. Take a look online at the "matplotlib gallery": `http://matplotlib.org/gallery.html`. This should give you some idea of the capabilities of matplotlib. This page very useful: clicking on a plot that shows something similar to what you want to create gives example code showing how that graph was created!

Another extremely useful library for physicists is the 'LINPACK' linear algebra package. This package provides very fast routines for calculating *anything* having to do with matrices: eigenvalues, eigenvectors, solutions of systems of linear equations, and so on. It's loaded under the name 'linalg' when you use ipython −−pylab.

Example 0.1.1
In electronics, Kirchhoff's laws are used to solve for the currents through components in circuit networks. Applying these laws gives us systems of linear equations, which can then be expressed as matrix equations, such as

$$\begin{bmatrix} -13 & 2 & 4 \\ 2 & -11 & 6 \\ 4 & 6 & -15 \end{bmatrix} \begin{bmatrix} I_A \\ I_B \\ I_C \end{bmatrix} = \begin{bmatrix} 5 \\ -10 \\ 5 \end{bmatrix} \tag{1}$$

This can be solved algebraically without too much difficulty, or one can simply solve it with LINPACK:

```
A = matrix([ [-13,2,4], [2,-11,6], [4,6,-15] ])
```

```
B = array([5,-10,5])
linalg.solve(A,B)
--> array([-0.28624535, 0.81040892, -0,08550186])
```

One can easily verify that the three values returned by linalg.solve() are the solutions for I_A, I_B, and I_C.

LINPACK can also provide eigenvalues and eigenvectors of matrices as well, using linalg.eig(). It should be noted that the size of the matrix that LINPACK can handle is limited only by the memory available on your computer.

0.2 Reading data from files

It's unlikely that you would be particularly excited by the prospect of manually typing in data from every experiment. The whole point of computers, after all, is to *save* us effort! Python can read data from text files quite well. We'll discuss this ability more in later in section 1.8, but for now here's a quick and dirty way of reading data files for graphing.

We'll start with a data file like that shown in table 1. This data file (which actually goes on for another three thousand lines) is from a lab experiment in another course at this university, and a copy has been provided[5]. Start iPython/pylab if it's not open already, and then use the loadtxt() function to read columns of data directly into Python variables:

Table 1: File microphones.txt

#Frequency	Mic 1	Mic 2
10.000	0.654	0.192
11.000	0.127	0.032
12.000	0.120	0.030
13.000	0.146	0.031
14.000	0.155	0.033
15.000	0.175	0.036
...		

[5] `/export/classes/phys312/examples/microphones.txt`

```
frequency, mic1, mic2 = loadtxt('microphones.txt', unpack = True)
```

The loadtxt() function takes one required argument: the file name. (You may need to adjust the file name (microphones.txt) to reflect the location of the actual file on your computer, or move the file to a more convenient location.) There are a number of optional arguments: one we're using here is "unpack", which tells loadtxt() that the file contains columns of data that should be returnend in separate arrays. In this case, we've told Python to call those arrays 'frequency', 'mic1', and 'mic2'. The loadtxt() function is very handy, and reasonably intelligent. By default, it will ignore any line that begins with '#', as it assumes that such lines are comments; and it will assume the columns are separated by tabs. By giving it different optional arguments you can tell it to only read certain rows, or use commas as delimiters, etc. It will choke, though, if the number of items in each row is not identical, or if there are items that it can't interpret as numbers.

Now that we've loaded the data, we can plot it as before:

```
figure()
plot(frequency, mic1, 'r-', frequency, mic2, 'b-')
xlabel('Frequency (Hz)')
ylabel('Amplitude (arbitrary units)')
legend(['Microphone 1', 'Microphone 2'])
```

See figure 2.

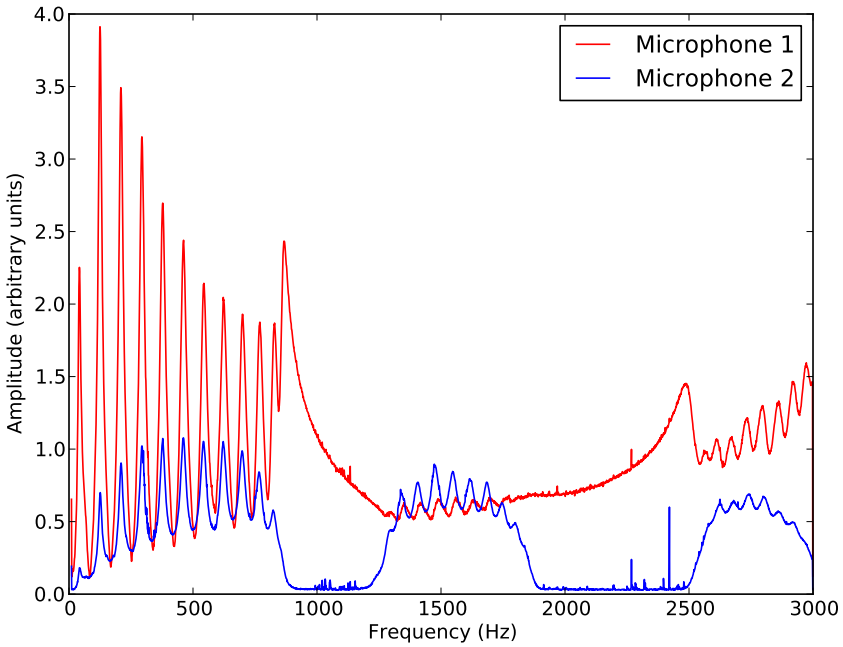


Figure 2: Data from 'microphones.txt'

0.3 Problems

0-0 Graph both of the following functions on a single figure, with a usefully-sized scale.

(a)
$$x^4 e^{-2x}$$

(b)
$$\left[x^2 e^{-x} sin(x^2)\right]^2$$

Make sure your figure has legend, range, title, axis labels, and so on.

0-1 The data shown in figure 2 is most usefully analyzed by looking at the *ratio* of the two microphone signals. Plot this ratio, with frequency on the x axis. Be sure to clean up the graph with appropriate scales, axes labels, and a title.

0-2 The file Ba137.txt contains two columns. The first is counts from a Geiger counter, the second is time in seconds.

(a) Make a useful graph of this data.

(b) If this data follows an exponential curve, then plotting the natural log of the data (or plotting the raw data on a logrithmic scale) will result in a straight line. Determine whether this is the case, and explain your conclusion with —you guessed it— an appropriate graph.

0-3 The data in file Ba137.txt is actual data from a radioactive decay experiment; the first column is the number of decays N, the second is the time t in seconds. We'd like to know the half-life $t_{1/2}$ of ^{137}Ba. It should follow the decay equation

$$N = N_o e^{-\lambda t}$$

where $\lambda = \frac{\log 2}{t_{1/2}}$. Using the techniques you've learned in this chapter, load the data from file Ba137.txt into appropriately-named variables in an ipython session. Experiment with different values of N and λ and plot the resulting equation on top of the data. (Python uses exp() calculate the exponential function: i.e. y = A∗exp(−L∗time)) Don't worry about automating this process yet (unless you *really* want to!) just try adjusting things by hand until the equation matches the data pretty well. What is your best estimate for $t_{1/2}$?

0-4 The normal modes and angular frequencies of those modes for a linear system of four coupled oscillators of mass m, separated by springs of equal strength k, are given by the eigenvectors and eigenvalues of M, shown below.

$$M = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

(The eigenvalues give the angular frequencies ω in units of $\sqrt{\frac{k}{m}}$.) Find those angular frequencies.

0-5 Create a single plot that shows separate graphs of position, velocity, and acceleration for an object in free-fall. Your plot should have a single horizontal time axis and separate stacked graphs showing position, velocity, and acceleration each on their own vertical axis. (See figure 3.) The online matplotlib gallery will probably be helpful! Print the graph, with your name in the title.

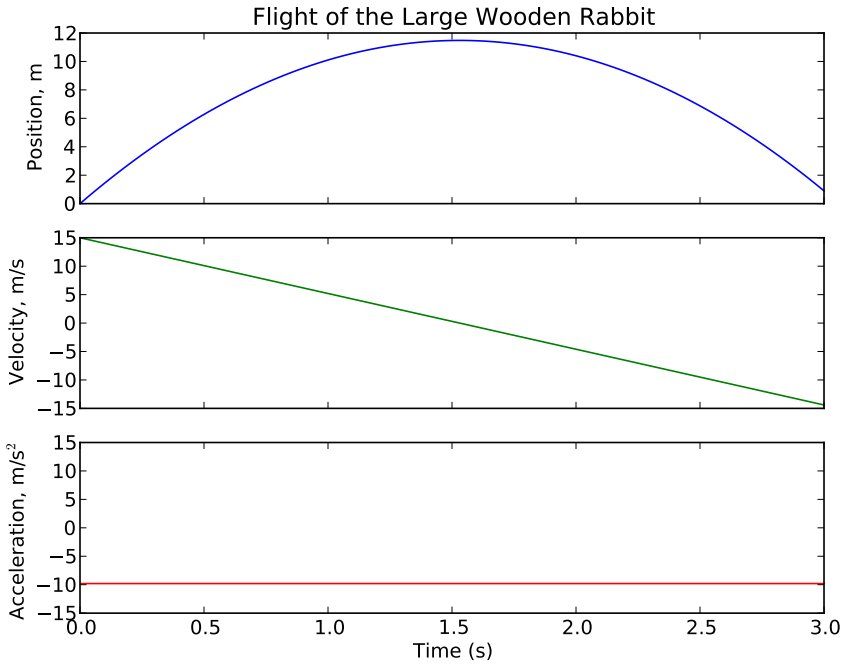


Figure 3: Sample three-graph plot

Chapter 1

Python Basics

1.0 The Python Interpreter

Python is a computer program which converts human-friendly commands into computer instructions. It is an *interpreter*. It's written in another language; most often C++, which is more powerful and much faster, but also harder to use.[1]

There is a fundamental difference between interpreted languages (Python, for example) and compiled languages such as C++. In a compiled language, all the instructions are analyzed and converted to machine code by the compiler *before* the program is run. Once this compilation process is finished, the program can run very fast. In an interpreted language, each command is analyzed and converted "on the fly". This process makes interpreted languages significantly slower; but the advantage to programming in interpreted languages is that they're easier to tweak and debug because you don't have to re-compile the program after every change.

Another benefit of an interpreted language is that one can experiment with simple Python commands by using the Python interpreter directly from the command line. In addition to the iPython method shown in the previous chapter, it's possible to use the Python interpreter directly. From a terminal window (Macintosh or Linux) type `python<enter>`, or open a window in "Idle" (Windows). You will get the Python prompt: `>>>`. Try some simple mathematical expressions, such as `6*7<enter>`. The Python interpreter takes each line of input you give it and attempts to make sense of it: if it can, it replies with what it got.

[1] There are versions of Python written in other languages, such as Java & C#. There is also a version of Python written in Python, which is somewhat disturbing.

You can use Python as a very powerful calculator if you want. It can also store value in variables. Try this:

```
x = 4
y = 16
x*y
x**y
y/x
x**y**x
```

That last one may take a moment or two: Python is actually calculating the value of $4^{(16^4)}$, which is a rather huge number.

In addition to taking commands one line at a time, the Python interpreter can take a file containing a list of commands, called a *program*. The rest of this book, and course, is about putting together programs so as to solve physics problems.

1.1 Comments

A program is a set of instructions that a computer can follow. As such, it has to be comprehensible by the computer, or it won't run at all. The rest of this chapter is concerned with the specifics of making the program comprehensible to the computer, but it's worthwhile to spend a little time here at the beginning to talk about making the program comprehensible to humans.

Python is pretty good in terms of comprehensibility. It's a language that doesn't require a lot of obscure punctuation or symbols that mean different things in different contexts. But there are still two very important things to keep in mind when you are writing any computer code:

(1) The next person to read the code will not know what you were thinking when you write the code.

(2) If you are the next person to read the code, rule #1 will still apply.

Because of this, it is absolutely critical to comment your code. Comments are bits of text in the program that the computer ignores. They are there solely for the benefit of any human readers.

Here's an example Python program:

```
#!/usr/bin/env python
"""
```

```
tenPrimes.py

    Here's a simple Python program to print the first 10
    prime numbers. It uses the function IsPrime(), which
    doesn't exist yet, so don't take the program too
    seriously until you write that function.
"""

# Initialize the prime counter
count = 0

# "number" is used for the number we're testing
# Start with 2, since it's the first prime.
number = 2

# Main loop to test each number
while count < 10:
    if IsPrime(number):      # The function IsPrime() should return
                             # a true/false value, depending on
                             # whether number is prime. This
                             # function is not built in, so we'd
                             # have to write it elsewhere.

        print number         # The number is prime, so print it.
        count = count + 1    # Add one to our count of primes so far.
    number = number + 1      # Add one to our number so we can check
                             # the next integer.
```

Anything that follows # is a comment. The computer ignores the comments, but they make the program easier for humans to read.

There is a second type of comment in that program also. Near the beginning there is a block of text delimited by three double-quotes: """. This is a multi-line string, which we'll talk more about later. The string doesn't *do* anything in this case, and isn't used for anything by the rest of the program, so Python promptly forgets it and it serves the same purpose as a comment. This specific type of comment is used by the `pydoc` program as documentation, so if you were to type the command `pydoc tenPrimes.py` the response would consist of that block of text. It is good practice to include such a comment at the beginning of each Python program. This comment should include a brief description of the program, instructions on how to use it, and the author & date.

There is one special comment at the beginning of the program:

#!/usr/bin/env python. This line is specific to Unix machines[2]. When the characters *#!* (called "hash-bang") appear as the first two characters in a file, Unix systems take what follows as an indicator of what the file is supposed to be. In this case, the file is supposed to be used by whatever the program `/usr/bin/env` considers to be the `python` environment.

Compare the program above with the following functionally identical program:

```
count = 0
number = 2
while count < 10:
    if IsPrime(number):
        print number
        count = count + 1
    number = number + 1
```

The second program might take less disk space but disk space is cheap and plentiful. Use the commented version.

1.2 Simple Input & Output

The raw_input() command takes user keystrokes and assigns them, as a raw string of characters, to a variable. The input() command does nearly the same, the only difference being that it first tries to make numeric sense of the characters. Either command can give a prompt string, if desired.

Example 1.2.1

```
name = raw_input("what is your name? ")
```

After the above line, the variable 'name' will contain the characters you type, whether they be "King Arthur of Britain" or "3.141592".

```
y = input("What is your quest? ")
```

The value of y, after you press enter, will be the computer's best guess as to the numeric value of your entry. "3.141592" would result in y being approximately π. "To find the Holy Grail" would cause an error.

[2]Including Macintosh & Linux, see appendix A

In order to get your carefully calculated results out of the computer and onto the monitor, you need the **print** command. This command sends the value of its arguments to the screen.

Example 1.2.2

```
e = 2.71828
print "Hello, world"
```

⟶ Hello world

```
print e
```

⟶ 2.71828

```
print "Euler's number is approximately ", e, "."
```

⟶ Euler's number is approximately 2.71828 .

Note in example 1.2.2 that the comma can be used to concatenate outputs. The comma can also be used to suppress the newline character that would otherwise come automatically at the end of the output. This use of the comma can allow you to make one print statement ending in a comma, then another print statement some lines further in the program, and have the output of both statements appear on one line of the screen.

It is also possible to specify the format of the output, using "string formatting". The most common format indicators used for our purposes are given in table 1.1. To use these format indicators, include them in an output string and then add a percent sign and the desired value to insert at the end of the print statement.

Example 1.2.3

```
pi = 3.141592
print "Decimal: %d" % pi
```

⟶ 3

```
print "Floating Point, two decimal places: %0.2f" % pi
```

⟶ 3.14

```
print "Scientific, two D.P, total width 10: %10.2e" % pi
```

%xd	Decimal (integer) value, with (optional) total width x.
%x.yf	Floating Point value, x wide with y decimal places.Note that the output will contain more than x characters if necessary to show y decimal places plus the decimal point.
%x.ye	Scientific notation, x wide with y decimal places.
%x.5g	"General" notation: switches between floating point and scientific as appropriate.
%xs	String of characters, with (optional) total width x.
+	A "+" character immediately after the % sign will force indication of the sign of the number, even if it is positive. Negative numbers will be indicated, regardless.

Table 1.1: Common string formatting indicators

$\longrightarrow$ 3.14e00

Note the two extra spaces at the front of the output in that final example. If we had given the format as "%2.2e", the output would have been the same numerically, but without those two blank spaces at the beginning. The output format expands as necessary, but always takes up *at least* as much space as specified.

Should you need to include more than one formatted variable in your output, go right ahead: just put the variables, grouped with parenthesis, after the % sign. Put them in the right order, of course: the first value will go into the first string formatting code, the second into the second, and so on.

Example 1.2.4

```
pi = 3.141592
e = 2.718282
sum = pi + e
print "The sum of %0.3f and %0.3f is %0.3f." % (pi, e, sum)
```

$\longrightarrow$ The sum of 3.142 and 2.718 is 5.860.

String formatting can be used for *any* strings, not just print statements. You can use string formatting to build up strings you want to use later, or send to a file, or whatever:

```
ComplicatedString = "Student %s scored %d on the final exam,
    for a grade of %s." % (name, FinalExamScore, FinalGrade)
```

1.3 Variables

It's worth our time to spend a bit of time discussing how Python handles variables. When Python interprets a line such as x=5, it starts from the right hand side and works its way towards the left. So given the statement x=5, the Python interpreter takes the "5", recognizes it as an integer, and stashes it in an integer-sized "box" in memory. It then takes the label x and uses it as a pointer to that memory location. (See figure 1.0.)

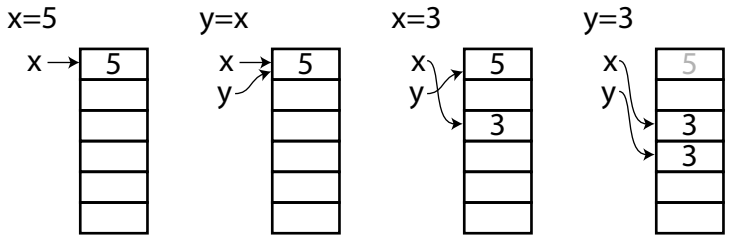


Figure 1.0: Variable assignment statements, and how Python handles them. The boxes represent locations in the computer's memory.

The statement y=x is analyzed the same way. Python starts from the right (x) and, recognizing x as a pointer to a memory location, makes y a pointer to the same memory location. At this point, both x and y are pointing to the same location in memory, and you could change the value of both *if you could change the contents of that location.* You can't, generally...

Continuing the example shown in figure 1.0, you now give Python the statement x=3. As always, this is analyzed from right to left: "3" is placed in some memory location, and x becomes a pointer to that location. This doesn't change y, though, since y is a pointer to the location containing "5".

Finally, if you now give Python the command y=3, y will end up pointing at some third memory location containing yet another "3". It will not be the same memory location as x, since from Python's perspective there is no reason it *should* be the same location. At this point, nothing is pointing at the memory location containing "5", and that location is free to be used for something else.

This right-to-left interpretation allows you to do some very useful —if mathematically improbable— things. For example, the command x = x+1 is perfectly legal in Python (as well as in nearly every other computer language.) If $x = 3$, as in the end of figure 1.0, Python would start from the right $(x + 1)$ and calculate that to be "4". It would then assign x to be a pointer to that "4". It is also perfectly legal in Python to say w = x = y = z = "Dead Parrot". In this case, each of those variables would end up pointing at the exact same spot in memory, until they were used for something else.

Python also allows you to assign more than one variable at a time. The statement a,b = 3,5 works because Python analyzes the right half first and sees it as a pair of numbers, then assigns that pair to the pair of variables on the left.[3] This can be used in some very handy ways.

Example 1.3.1
You want to swap two variable values.

```
x,y = y,x
```

Example 1.3.2
If you start with $a = b = 1$, what would be the result of repeated uses of this command?

```
a,b = b, a+b
```

Generally, a deep knowledge of how Python manages variables like this is not necessary. But there are occasions when changing a variable's value changes the contents of that box in memory rather than changing the address pointed to by the variable. Keep this in mind when you're dealing with matrices. It's something to be aware of!

Variable Names

Variable names can contain letters, numbers, and the underscore character. They must start with a letter. Names are case sensitive, so "Time" is not the same as "time".

[3] Technically, it sees both pairs as "tuples". See page 22.

It's good practice when naming variables to choose your names so that the code is "self-commenting". The variable names r and R are legal, and someone reading your computer code might guess that they refer to radii; but the names CylinderRadius and SphereRadius are much better. The extra time you spend typing those more descriptive variable names will be *more* than made up by the time you save debugging your code!

Variable Types

There are many different types of variables in Python. The two broad divisions in these types are *numeric* types and *sequence* types. Numeric types hold single numbers, such as "42", "3.1415", and $2 - 3i$. Sequence types hold multiple objects, which may be single numbers, or individual characters, or even collections of different types of things.

One of the strengths (and pitfalls) of Python is that it automatically converts between types as necessary, if possible.

Numeric Types

Integer The integer is the simplest numeric type in Python. Integers are perfect for counting items, or keeping track of how often you've done something.

The maximum integer is $2^{31} - 1 = 2,147,483,647$.

Integers don't divide quite like you'd expect, though! In Python, $1/2 = 0$, because 2 goes into 1 zero times.

Long Integer Integers larger than 2,147,483,647 are stored, automatically, as long integers. These are indicated by a trailing "L" when you print them, unless you use string formatting to remove it.

The maximum size of a long integer is limited only by the memory in your computer. Integers will automatically convert to long integers if necessary.

Float The "floating point" type is a number containing a decimal point. 2.718, 3.14159, and 6.626×10^{-34} are all floating point numbers. So is 3.0. Floats require more memory to store than do integers, and they are generally slower in calculations, but at least $1.0/2.0 = 0.5$ as one would expect.

It's important to note that Python will "upconvert" types if necessary. So, for example, if you tell Python to calculate the value of 1.0/2,

Python will convert the integer 2 to the float 2.0 and then do the math.There is a trade-off in speed for this convenience, though.

Complex Complex numbers are built-in in Python, which uses $j \equiv \sqrt{-1}$. It is perfectly legal to say x = 0.5 + 1.2j in Python, and it does complex arithmetic correctly.

Sequence Types

Sequence types in Python are collections of items which are referred to by one variable name. Individual items within the sequence are separated by commas, and referred to by an *index* in square brackets after the sequence name. This is easier to demonstrate than explain, so...

Example 1.3.3

```
Pythons = ("Cleese", "Palin", "Idle", "Chapman",
    "Jones", "Gilliam")
print Pythons[2]
```

⟶ Idle

Note that the index starts counting from zero:

```
print Pythons[0]
```

⟶ Cleese

Negative numbers start counting from the end, backwards:

```
print Pythons[-1], Pythons[-2]
```

⟶ Gilliam Jones

One can also specify a "slice" of the sequence:

```
print Pythons[1:3]
```

⟶ ('Palin', 'Idle')

Note that in that last example, what is printed is another (shorter) sequence. Note also that the range [1:3] tells Python to start with item 1 and go *up to* item 3. Item 3 is not included.

Now let's examine some of the specific types of sequence in Python.

Tuple Tuples are indicated by parentheses: (). Items in tuples can be any other data type, including other tuples. Tuples are *immutable*, meaning that once defined their contents cannot change.

List Lists are indicated by square brackets: [ ]. Lists are pretty much the same as tuples, but they are *mutable*: individual items in a list may be changed. Lists can contain any other data type, including other lists.

String A string is a sequence of characters. Strings are delimited by either single or double quotes: " " or ' '. Strings are immutable, like tuples. Unlike lists or tuples, strings can only include characters.

There are also some special characters in strings. To indicate a <tab> character, use "\t". For a newline character, use "\n".

The # character indicates a comment in Python, so if you put # in a string the rest of the string will be ignored by the Python interpreter. The way to get around this is to "escape" the character with "\", such as "\#". This causes Python to recognize that the # is meant as just a # character, rather than the *meaning* of #. Similarly, \" will allow you to put a double-quote inside a double-quoted string, and \' will allow use of a single-quote inside a single-quoted string.

An alternate way of indicating strings is to bracket them in triple double-quotes. This allows you to have a string that spans multiple lines, including tabs and other special chacters. A triple double-quoted string within a Python program which is not assigned to a variable or otherwise used by Python will be taken to be documentation by the pydoc program.

Dictionary Dictionaries are indicated by curly brackets: { }. They are different from the other built-in sequence types in Python in that instead of numeric indices they use "keys", which are string labels. Dictionaries allow some very powerful coding, but we won't be using them in this course so you'll have to learn them elsewhere.

As mentioned in the description of lists above, a list can contain other lists. A list of lists sounds almost like a 2-dimensional array, or matrix. You can use them as such, and the way you would refer to elements in the matrix is to tack indices onto the end of the previous index.

Example 1.3.4

```
matrix = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

The variable "matrix" is now a 3×3 array. To change the first item in the second row from 4 to 0, we would use

```
matrix[1][0] = 0
```

(Remember that the indices start from zero!)

This is almost what we want for matrices in computational physics. Almost. The [i][j] method of indexing is somewhat clumsy, for starters: it'd be nice to use [i,j] notation like we do in everything else. Another problem with using lists of lists for matrices is that addition doesn't work like we'd expect. When lists or tuples or strings are added, Python just sticks them together end-to-end, which is mathematically useless.

Example 1.3.5

```
matrix1 = [ [1, 2], [3, 4] ]
matrix2 = [ [0, 0], [1, 1] ]
print matrix1 + matrix2
```

$\longrightarrow$ [[1, 2], [3, 4] [0, 0], [1, 1]]

The best way of doing matrices in Python is to use the SciPy or NumPy packages, which we will introduce later.

Sequence Tricks

If you are calculating a list of N numbers, it's often handy to have the list exist first, and then fill it with numbers as you do the calculation. One easy way to create an empty list with the length needed is multiplication:

```
LongList = []*N
```

After the above command, LongList will be a list of N blank elements, which you can refer to as you figure out what those elements should be.

Sometimes you may not know exactly how many list elements you need until you've done the calculation, though. Being able to add elements to the end of a list, thus making the list longer, would be ideal in this case; and Python provides for this with list.append(). Here's an example:

Example 1.3.6

You want a list of calculated values, but you don't know exactly how many you need ahead of time.

```
# Start by creating the first list element.
# Even an empty element will do -- there just must
# be something to tell Python that the variable
# is a list rather than something else.
Values = []
# Now do your calculations,
NewValue = MuchCalculation(YadaYadaYada)
# and each time you find another value just append
# it to the list:
Values.append(NewValue)
# This will increase the length of Values[] by one,
# and the new element at the end of Values[] will
# be NewValue.
```

Another handy trick is sorting. If, for example, in the previous example you wanted to sort your list of values numerically after you'd calculated them all, this would do it:

```
Values.sort()
```

After that, the list Values would contain the same information but in numeric order.

Generally speaking, the sort operation only works if all the elements of the list are the same type. Trying to sort a mix of numbers and strings and other sequences is a recipe for disaster, and Python will just give you an error and stop.

Sequences have many other useful built-in operations, more than can be covered here. Google is my preferred way of finding what they are: If there's something you think you *should* be able to do with a sequence, Google "Python list (or string, or tuple) <action>", whatever that action might be. This will usually come up with something! There are operations for finding substrings in strings, changing case in strings, removing non-printing characters, etc...

Ranges

It is often necessary to create a list of numbers for the computer to use. If you're making a graph, for example, it'd be nice to quickly generate a list of numbers to put along the bottom axis. Python has a built-in function to do this for us: "range()". The range() function takes up to three parameters: *Start* (optional), *Stop* (required), and *Step* (optional). It generates a list of

integers beginning with *Start* (or zero if *Start* is omitted) and ending just before *Stop*, incrementing by *Step* along the way.

Example 1.3.7
Create a list of 100 numbers for a graph axis.

```
axis = range(100)
print axis
```

⟶ [0, 1, 2, ... 98, 99]

Create a list of even numbers from 6 up to 17.

```
Evens = range(6,17,2)
```

List Comprehensions

The range() function only creates integers, and the spacing between the integers is always exactly the same. We often need something more flexible than that: what if we needed a list of squares of the first 100 integers, or a range that went up by 0.1 each step?

Python provides one very useful trick for doing just this thing: List Comprehensions. Again, this is most easily shown by example:

Example 1.3.8
Create a list of 100 evenly-spaced numbers for a graph axis which goes from a minimum of zero to a maximum of 2.

```
Axis = [0.02 * i for i in range(100)]
```

Let's look at that bit by bit. The square brackets indicate that the result is a list. The formuala (0.02 * i) is how it calculates each item in the list. The "for i in range(100)" tells Python to take each number in the list "range(100)", call that number "i", and apply the formula given.

List comprehensions work for *any* list, not just range(). So if you have a list of experimental measurements that were taken in inches, and you needed to convert them all to centimeters, then the line

```
metric = [2.54*measure for measure in ListOfMeasurements]
```

would do what you need.

1.4 Mathematical Operators

In Python, the mathematical operators $+ - * ()$ all work as one would expect on numbers and numeric variables. As mentioned previously, the + operator doesn't do matrix addition on lists — it just strings them together instead. The * operator (multiplication) also does something unexpected on lists: if you multiply a list by n, the result will be n copies of the list, strung together. As long as you stick with numeric types, though, addition, subtraction, and multiplication do what you want.

Division (/) is slightly different. It works perfectly on floats, but on integers it does "third-grade math": the result is always the integer portion of the actual answer.

Example 1.4.1

```
print 10/4
```

$\longrightarrow 2$

```
print 1/2
```

$\longrightarrow 0$

```
print 1./2.
```

$\longrightarrow 0.5$

That second case in example 1.4.1 is particularly bothersome: it causes more program bugs than you'd expect, so keep an eye out for it.

If you *want* "third-grade math" with floats, use the "floor division" operator, //.

Exponentiation in Python is done with the ** operator, as in

```
Sixteen = 2**4
```

The modulo operator (AKA "remainder") is %, so $10\%4 = 2$ and $11\%4 = 3$.

The precedence rules in Python are exactly what they should be in any mathematics system: (); then **; then *, /, %, // in left-to-right order; and finally +, − in left-to-right order. Although those are the rules, and they work, the *best* way to do things is to use the "cheap rule": "* before +, otherwise use ()."

Shortcut Operators

The statement x = x + y and other similar statements are so common in programming that many languages (including Python) allow shortcut operators for these statements. The most common of these is +=, which means "Take what's to the right and add to it whatever is on the left". In other words, x += 1 is exactly equivalent to x = x + 1. Similarly, − =, ∗ =, and / = do the same thing for subtraction, multiplication, and division.

These shortcut operators do not make your program run faster, and they do make the code harder for humans to read. Use them sparingly.

1.5 Lines in Python

Python is somewhat unique among programming languages in that it is whitespace-delimited. In other words, the Python interpreter actually cares about blank space before commands on a line.[4] Lines are actually grouped by how much whitespace preceeds them, which forces one to write well-indented code!

When one speaks of "lines" of Python code, there are actually two types of line. A *physical* line is a line that takes up one line on the editor screen. A *logical* line is more important — it's what the Python interpreter regards as one line. Let's look at some examples:

Example 1.5.1

```
print "this line is a physical line and a logical line."

x = ["this", "line", "is", "both", "also"]

x = ["this", "line", "is", "multiple",
     "physical", "lines", "but", "is",
     "just", "one", "logical", "line"]
```

Indentation like this helps make programs clear and easier for humans to read. Python ignores extra whitespace *inside* a logical line, so it's not a problem to put it there.

---

[4] Whether this is one of the best or worst features of Python is a matter of some debate.

1.6 Control Structures

Control statements are statements that allow a program to do different things depending on what happens. "If you are hungry, eat lunch." is a control statement of sorts. "While you are in Hawaii, enjoy the beach." is another. Of course control statements in a computer language are a bit more specific than that, but they have the same basic structure. There is the statement itself: "if". There is the "conditional", which is a statement that evaluates to either true or false: "you are hungry". And there is the action: "eat lunch".

Conditionals

A conditional is anything that can be evaluated as either true or false. In Python, the following things are always false:

- The word False.
- 0, 0L, or 0.0
- "" or '' (an empty string)
- (), [], {} (The empty tuple, list, or dictionary)

Just about everything else is true.

- 1, 3.14, 42 (True, because they are numbers that aren't zero)
- The word True.
- "False" (This is true, because it's a string that is not empty. Python doesn't look inside the string to see what it's about!)
- "0", for the same reason as "false".
- [0, False, (), ""] (This is true, even though it's a list of false things, because it is a non-empty list, and if a list is not empty then it is true.)

The comparisons are

< Less than

> Greater than

<= Less than or equal to

>= Greater than or equal to

== Equal to

!= Not equal to

Note that "=" is an assignment, and "==" is a conditional. FavoriteColor = ''Blue'' assigns the string "Blue" to the variable FavoriteColor, but FavoriteColor == ''Blue'' is a conditional that evaluates to either true or false depending on the contents of the variable FavoriteColor. *Using = instead of == is one of the most common and hard-to-find bugs in Python programs!*

There are also the boolean operators **and**, **or**, and **not**.

Example 1.6.1

```
if (Animal == ''Parrot'') and (not IsAlive(Animal)):
    Complain()
```

The precedence of **and**, **or**, and **not** is the lowest of anything in Python, so the parentheses are not actually necessary. Those parentheses make the code more readable, though, and as such are highly recommended.

One final boolean is the **in** command, which is used to test whether an item is in a list.

```
Cast = ('John', 'Eric', 'Terry', 'Graham', 'Terry', 'Michael')
if Name in Cast:
    print 'Yes', Name, "is a member of Monty Python's Flying Circus"
```

If . . . Elif . . . Else

The most basic control statement in Python, or any other computer language, is the **if** statement. It allows you to tell the computer what to do if some condition is met. The syntax is as follows:

```
if Conditional: # The : is needed at the end of this line.
    DoThis()    # This line must be indented.
    AndThis()   # Any other lines that are part of the if
                # statement must also be indented the same
                # amount. (comments are ignored, of course!)
    ThisAlso()
AlwaysDoThis()  # This line is not indented, so it is not
                # part of the if statement's action.
```

Note the indenting. In other computer languages, indentation like this makes the code easier to read; but in Python *the indentation is what defines the group of commands.* The indentation is *not* optional.

The **elif** and **else** keywords extend the **if** statement even further. **elif**, short for "else if" adds another if that is tested only if the first if is false. **else** is done if none of the previous **elif** statements, or the initial **if**, are true.

```
If TestOne:
    Do_A()          # Do_A() is done only if TestOne is true.
elif TestTwo:       # TestTwo is tested only if TestOne is false.
    Do_B()          # Do_B() is done if TestOne is false and
                    # TestTwo is true.
elif TestThree:
    Do_C()          # You can have as many elif's as you want,
                    # or none at all. They're optional.
else:
    Do_D()          # The 'else' is what happens if nothing else is true.

AlwaysDoThis()      # This statement is back at the left margin again.
                    # That means it's after the end of the whole if
                    # construct, and it is done regardless.
```

While

The **while** statement is used to repeat a block of commands until a condition is met.

```
while Condition:        # The : is required, again.
    DoThis()            # The block of things that should be done
    DoThat()            # is indented, as always.
    DoTheOther()
    UpdateCondition()
                        # If nothing happens to change the condition,
                        # the while loop will go forever!
DoThisAfterwards()      # This statement is done after Condition
                        # becomes false.
```

There are a few extra keywords that can be used with the **while** loop.

pass The **pass** keyword does exactly nothing. Its sole purpose is to create an indented line if you want the program to do nothing but there's a structural need for an indented line. I know this seems crazy, but I have actually found a situation in which the **pass** command was the simplest way to make something work!

continue The **continue** keyword moves program execution to the top of the while block without finishing the portion of the block following the **continue** statement.

break The **break** keyword moves execution of the loop directly to the line following the **while** block. It "breaks out" of the loop, in other words.

else An **else** command at the end of a **while** block is used to delineate a block of code that is done after the **while** block executes *normally*. The code in this block is *not* executed if the **while** block is exited via a **break** command.

Confusing enough for you? This is probably a good time for an example that uses these features.

Example 1.6.2
You need to write a program that tests whether a number is prime or not. The program should ask for the integer to test, then print a message giving either the first factor found or stating that the number is prime.

```
#!/usr/bin/env python
"""
    This Python program determines whether a number
    is prime or not. It is NOT the most efficient
    way of determining whether a large integer is
    prime!
"""

# Start by getting the number that might be prime.
Number = input("What integer do you want to check? ")

TestNumber = 2
# Main loop to test each number
while TestNumber < Number:
    if Number % TestNumber == 0:
        # The remainder is zero, so TestNumber is a factor
        # of Number and Number is not prime.
        print Number, "is divisible by", TestNumber, "."
        # There is no need to test further factors.
        break
    else:
        # The remainder is NOT zero, so increment
        # TestNumber to check the next possible factor.
```

```
            TestNumber += 1
    else:
        # We got here without finding a factor, so
        # Number is prime.
        print Number, "is prime."
```

For

The **for** loop iterates over items in a sequence, repeating the loop block once per item. The most basic syntax is as follows:

```
for Item in Sequence:    # The : is required
    DoThis()             # The block of commands that should
    DoThat()             # be done repeatedly is indented.
```

Each time through the loop, the value of Item will be the value of the next element in the Sequence. There is nothing special about the names Item and Sequence, they can be whatever variable names you want to use. In the case of Sequence, you can even use something that *generates* a sequence rather than a sequence, such as range().

Example 1.6.3
You need a program to greet the cast of a humorous skit.

```
Cast = ('John', 'Eric', 'Terry', 'Graham', 'Terry', 'Michael')
for Member in Cast:
    print 'Hello', member            # Each time through the loop
                                     # one cast member is greeted.
print 'Thank you for coming today!' # This line is outside the loop
                                     # so it is done once, after
                                     # the loop.
```

The output of this program will be

```
Hello John
Hello Eric
Hello Terry
Hello Graham
Hello Terry
Hello Michael
Thank you for coming today!
```

In numeric work, it's more common to use the **for** command over a numeric range:

```
for j in range(N):
    etc()
```

Since the range() function returns a list, the **for** command is perfectly happy with that arrangement.

The **for** command also allows the same extras as **while**: **continue** goes straight to the next iteration of the **for** loop, **break** causes Python to abandon the loop, **else** at the end of the loop marks code that is done *only* if the **for** loop exits normally, and **pass** does nothing at all.

One caution about **for** loops: it's quite possible to change the list that is controlling the loop during the loop! This is not recommended, as the results may be unpredictable.

1.7 Functions

A function is a bit of code that is given its own name so that it may be used repeatedly by various parts of a program. A function might be a bit of mathematical calculation, such as sin() or sqrt(). It might also be code to do something, such as draw a graph or save a list of numbers.

Functions are defined with the **def** command. The function name must start with a leter, and may contain letters, numbers, and the underscore character just like any other Python variable name. In parentheses after the function name should be a list of variables that should be passed to the function. The **def** line should end with a colon, and the indented block after the **def** should contain the function code.

Generally, a function should **return** some value, although this is not required. For mathematical functions, the return value should be the result of the calculation. For functions that don't calculate a mathematical value, the return should be True or False depending on whether the function managed to do what it was supposed to do or not.

Example 1.7.1
Write a function that calculates the factoral of a positive integer.

```
def factoral(n):
    """ factoral(n)
        This function calculates n! by the simplest and
```

```
        most direct method imaginable.
    """
    f = 1
    for i in range(2, n+1):
        f = f * i
    return f
```

Once this definition has been made, any time you need to know the value of a factorial, you can just use factoral(x).

```
print '%10s %10s' % ('n', 'n!')
for j in range(10):
    print '%10d %10d' % (j, factoral(j))
```

Functions can also be used to break the code up into more understandable chunks. Your morning ritual might look something like this, in Python code:

```
if (Time >= Morning):
    GetUp()
    GetDressed()
    EatBreakfast(Spam, eggs, Spam, Spam, Spam, Spam, bacon)
else:
    ContinueSleeping()
```

The functions GetDressed() and EatBreakfast() may entail quite a bit of code; but writing them as separate functions allows one to bury the details (socks first, then shoes) elsewhere in the program so as to make this code more readable. Writing the program as a set of functions also allows you to change the program easily, if for example you needed to eat breakfast before getting dressed.

The variables that are passed to the function exist for the duration of that function only. They may (or may not) have the same name(s) as other variables elsewhere in the program.

Example 1.7.2

```
def sq(x):
    # Returns the square of a number x
    x = x*x # Note that sq() changes the local value of x here!
    return x

# Here's the main program:
x = 3
```

```
print sq(x) # prints 9,
print x     # prints 3.
```

Note that the value of x is changed within the function sq(), but that change doesn't "stick". The reason for this is that sq() does not receive x, but instead receives some value which it then calls x for the duration of the function. The x within the function is not the same as the x in the main program.

Functions can have default values built-in, which is often handy. This is done by putting the value directly into the definition line, like this:

```
def answer(A = 42):
    # Put your function here
    # etc.

# main program
answer(6)    # For this call to answer(), A will be 6.
answer()     # But this time, A will be the default, 42.
```

Global variables

If a Python function can't find the value of some variable, it looks outside the function. This is handy: you can define π once at the beginning of the program and then use it inside any functions in the program. Values used throughout the program like this are called global variables.

If you re-define the value of a variable inside your function, though, then that new value is valid only within the function. To change the value of a global variable and make it stick outside the function, refer to that variable in the function as a **global**. The following example may help clarify this.

Example 1.7.3

```
a = 4
b = 5
c = 6

def fn(a):
    d = a           # d is a new local variable, and has the
                    # value of whatever was passed to fn().
    a = b           # 'fn' does not know 'b', so Python looks
                    # outside fn and finds b=5. So the local
```

```
                    # value of 'a' is 5 now.
        global c = 9 # Instead of making a new local 'c', this
                    # line changes the value of the global
                    # variable 'c'.

    print a,b,c     # --> 4 5 6
    fn(b)           # The value of d inside fn() will be 5.
                    # The value of a inside fn() will also be 5.
    print a,b,c     # --> 4 5 9
                    # The values outside fn() didn't change,
                    # other than c.
    print d         # --> ERROR! d is only defined inside fn().
```

Passing functions

Python treats functions just like any other variable. This means that you can store functions in other variables or sequences, and even pass those functions to other functions. The following program is somewhat contrived, but it serves to give a good example of how this can be useful. The output is shown in figure 1.1.

```
#!/usr/bin/env python
''' passtrig.py
    Demonstrates Python's ability to store functions an variables and
    pass those functions to other functions.
'''

from pylab import *

def plottrig(f):
    # this function takes one argument, which must be a function
    # to be plotted on the range -pi..pi.
    xvalues = linspace(-pi, pi, 100)
    plot(xvalues, f(xvalues))   # y values are computed depending on f.
    xlim(-pi, pi)               # set x limits to x range
    ylim(-2, 2)                 # set the y limits so that tan(x) doesn't
                                # ruin the vertical scale.

trigfunctions = (sin, cos, tan) # trigfunctions is now a tuple that holds
                                # these three functions.

for function in trigfunctions:
    # trigfunctions is a list of functions, so this for loop does things
```

```
    # with each function in the list.
    print function(pi/6.0)        # returns this function value.
    plottrig(function)            # passes this function to be plotted.

show()
```

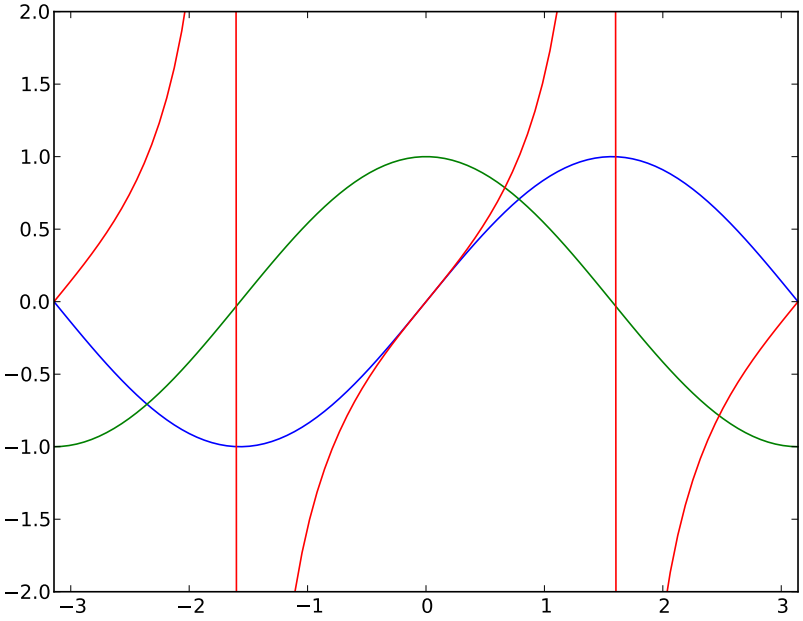


Figure 1.1: Output of function-passing program "passtrig.py"

As you can see, functions here are put into lists, referred to as elements in lists, and passed to other functions. This is particularly useful in the next chapter, where we will develop (among other things) ways of finding roots of equations. We can write functions that find roots, then pass functions to be solved *to* those root-finding functions. This allows us to use one general-purpose root-finding function for just about any function for which we need to find a root.

1.8 Files

More often than not in computational physics, the inputs and outputs of a project are large sets of data. Rather than re-enter these large data sets each time we run the program, we load and save the data in the form of text files.

When working with files, we start by "opening" the file. the open() function tells the computer operating system what file we will be working on, and what we want to do with the file. The function requires two parameters: the file name and the desired mode.

```
FileHandle = open(FileName, Mode)
```

FileName should be a string describing the location and name of the file, including any directory information if the file is not in the working directory for the Python program. The mode can be one of three things:

'r' Read mode allows you to read the file. You can't change it, only read it.

'w' Write mode will create the file if it does not exist. If the file *does* exist, opening it in 'w' mode will re-write it, destroying the current contents.

'a' Append mode allows you to write onto the end of a previously-existing file without destroying what was there already.

Once the file is open for reading, we can read it by one of several methods. We can read the entire text file into one string:

```
string = FileHandle.read()
```

The resulting string can be inconveniently huge, though, depending on the file! We could alternately read one line of the file at a time:

```
Line = FileHandle.readline()
```

Each time you invoke this command, the variable Line will become a string containing the next line of the file indicated by FileHandle. We could also read all of the lines at once into a list:

```
Lines = FileHandle.readlines()
```

After this command, Lines[0] will be a string containing the first line of the file, Lines[1] will contain the second line, and so on.

When we're done reading the file, it's best to close the file.

```
FileHandle.close()
```

If you don't close the file, it will close automatically when the program is done, but it's still good practice to close it yourself.

Notice that all of these methods of reading a file result in strings of characters. This is to be expected, since the file itself is a string of characters on the drive. Python makes no effort to try figuring out what those characters mean, so if you want to change the strings to numeric values you must convert them yourself by specifying what type of numbers you expect them to be. The float() and int() functions are the standard way of doing this. For example if the string S = '3.1415', then x=float(S) takes that string and changes it to the appropriate floating-point value.

If you wish to write to a file, you must indicate so when you open the file.

```
FileHandle = open(FileName, 'w')
```

After the file is open for writing, the write operation saves string information to the file:

```
FileHandle.write(string)
```

The write() operation is much more literal than the **print** command: it sends to the file *exactly* the contents of the string. If the string does not have a newline character at the end, then there will be no newline character written to the file, so if you want to write a single line be sure to include '\n' as the last character in the string. The write() operation can use any of the string formatting techniques described earlier.

```
FileHandle.write("pi = %6.4f\te=%6.4f\n" % (pi, e))
```

When you are done writing to the file, it is very important to close the file.

```
FileHandle.close()
```

Closing the file flushes the write buffer and ensures that all of the written matter is actually on the hard drive. The file will be closed automatically at the end of the program, but it's better to close it yourself.

1.9 Expanding Python

One of the nicest things about Python is how easy it is to add further functionality. In the mathematical department, for example, Python is somewhat limited. It does not have built-in trigonometric functions, or know the values of π and e. But this can be easily added into Python, as needed, using the **import** command.

```
import math
```

import commands are generally placed at the beginning of the program, although that's just for convenience to the reader. Once the **import** command has been run by the program, all of the functions in the "math" package are available as "math.(function-name)".

```
import math
x = math.pi / 3.0
print x, math.sin(x), math.cos(x), math.tan(x)
```

There are many other functions and constants in the math package.[5] It is often useful to just import individual elements of a package rather than the entire package, so that we could refer to "sin(x)" rather than math.sin(x).

```
from math import sin
```

You can also import everything from a package, each with its own name:

```
from math import *
x = pi / 3.0
print x, sin(x), cos(x), tan(x)
```

There are numerous other packages that add useful functionality to Python. In this book we'll be using the "scipy", "pylab", "matplotlib", and "numpy" packages extensively. These will be discussed later, but one package that deserves mention here is the "sys" package, which allows access to some of the system underpinnings of a unix-based system. The part of the sys package we use most often is the variable sys.argv. When you invoke a Python program from the command line, it is possible to enter parameters directly at time of invocation.

```
$ add.py 35 7
```

In the case shown above, the variable sys.argv will be a list containing the program name and whatever came after on the line:

```
sys.argv = ['add.py', '35', '7']
```

So we can use this to enter values directly into a program.

Example 1.9.1

```
import sys

x = float(sys.argv[1])   # Note that the items in argv[]
y = float(sys.argv[2])   # are strings!

print "%0.3f + %0.3f = %0.3f" % (x, y, (x+y))
```

[5] Give the command "pydoc math" in a terminal window for more information.

If this program were saved as "add.py", a user command of
`add.py 35 7`
would result in an output of
`35.000 + 7.000 = 42.000`

Note that the components of sys.argv are strings, so it's necessary to convert them to floating-point numbers with the float() function so that they add as numbers.

You can also build your own packages. This is easier than you might imagine, since *every* Python program is a package already! Any Python program ending with a '.py' suffix on its filename (and they all should end with .py) can be imported by any other Python program. Just use **from** <filename> **import** <function(s)>.

Example 1.9.2
You've written a program called "area.py" that includes an integration routine "integrate()", and you'd like to use that routine in your next program.

```
from area import integrate
```

It's a good idea, as you work through the exercises in this course, to put any useful functions you may develop into a "tools.py" file. Then all of those functions are easily accessible from new programs with the command

```
from tools import *
```

One thing to consider, though: any time you import a package, it overwrites anything with the same name. This can cause problems. The pylab package has trig functions, for example, that are capable of operating on entire arrays at once. The math package trig functions have much more basic capabilities. So if you use this code:

```
from pylab import *
from math import *
x = sin([0.1, 0.2, 0.3, 0.4, 0.5])
```

then the basic trig functions from math.py overwrite the pylab.py trig functions, and the sin() function will crash when you give it the array.

A better way to do it, if you *must* have both pylab and math packages, would be to import the packages with their "full names" intact.

```
import pylab
import math
x = pylab.sin([0.1, 0.2, 0.3, 0.4, 0.5])
y = math.sin(pi/2)
```

There's another thing to consider when dealing with Python packages: *any* Python program can be imported as a package into any other Python program. A Python program can even import itself, which will cause nothing but trouble. Problems arise if you happen to name your program "math.py" and them import math, for example. Python will happily import the first "math.py" it finds, which will not be the one you expect and nothing will work right and the error messages you get will be completely unhelpful.

1.10 Where to go from Here

This chapter has barely scratched the surface of the Python language, but it hopefully gives you enough of a foundation in Python that you'll be able to get started on solving some interesting physics problems.

There's a lot more to learn. One excellent resource for learning more Python is the book *Learning Python* by Mark Lutz and David Ascher.[6] Another is *Computational Physics* by Mark Newman.[9] The official Python website is at `http://www.python.org/`, and it always has up-to-date documentation about the latest version of Python. Google and other search engines are also excellent resources: there are numerous pages with helpful descriptions about how to do things in Python.

One other thing to be aware of: This chapter (this entire book, actually) is based on Python 2.7. Python 3.0 was released in November of 2008, and 3.0 is not entirely reverse-compatible with 2.7! Although 2.7 is not the latest-and-greatest, not all of the packages we use in the rest of this text are available for 3.x at this time. It is my hope that the matplotlib, visual, and numpy packages will be updated to work with 3.x soon, and until then 2.7 is the preferred version of Python for numerical work.

1.11 Problems

1-0 Use a list comprehension to create a list of squares of the numbers between 10 and 20, including the endpoints.

1-1 Write a Python program to print out the first N numbers in the Fibonacci sequence. The program should ask the user for N, and should require that N be greater than 2.

1-2 For the model used in introductory physics courses, a projectile thrown vertically at some initial velocity v_i has position $y(t) = y_i + v_i t - \frac{1}{2}gt^2$, where $g = 9.8$ m/s^2. Write a Python program that creates two lists, one containing time data (50 data points over 5 seconds) and the other containing the corresponding vertical position data for this projectile. The program should ask the user for the initial height y_i and initial velocity v_i, and should print a nicely-formatted table of the list values after it has calculated them.

1-3 The energy levels for a quantum particle in a three-dimensional rectangular box of dimensions $\{L_1, L_2, \text{and } L_3\}$ are given by

$$E_{n_1,n_2,n_3} = \frac{\hbar^2\pi^2}{2m}\left[\frac{n_1^2}{L_1^2} + \frac{n_2^2}{L_2^2} + \frac{n_3^2}{L_3^2}\right]$$

where the n's are integers greater than or equal to one. Write a program that will calculate, and list in order of increasing energy, the values of the n's for the 10 lowest *different* energy levels, given a box for which $L_2 = 2L_1$ and $L_3 = 4L_1$.

1-4 Write a function for

$$\text{sinc}(x) \equiv \frac{\sin x}{x}$$

Make sure that your function handles $x = 0$ correctly.

1-5 Write a function that calculates the value of the n^{th} triangular number. Triangular numbers are formed by adding a series of integers from 1 to n: i.e. $1 + 2 + 3 + 4 = 10$ so 10 is a triangular number, and 15 is the next triangular number after 10.

1-6 Write a function called `isprime()` that determines whether a number is prime or not, and returns either `True` or `False` accordingly. Redo Example 1.6.2 using this function. Try to make your program more efficient than the example, which is pretty dreadful in that regard!

1-7 Write a Python program to make an $N \times N$ multiplication table and write this table to a file. Each row in the table should be a single line of the file, and the numbers in the row should be tab-delimited. Write the program so that it accepts two parameters when invoked: the size N of the multiplication table and the filename of the desired output file. In other words, the command to make a 5×5 table saved in file table5.txt would be

```
timestable.py 5 table5.txt
```

1-8 Write a program that takes a multi-digit integer and prints out its digits in English. Bonus point if the program takes care of ones/ tens/hundreds/thousands properly.

Chapter 2

Basic Numerical Tools

2.0 Numeric Solution

A problem commonly given in first-semester physics lab is to calculate the launch angle for a projectle launcher so that the projectile hits a desired target. This is an easy enough problem, if the launch point and the target are at the same elevation: you just take the range equation

$$R = \frac{v^2 \sin(2\theta)}{g}$$

and solve for θ.

Complications arise, though, when the launcher is on the table and the target is on the floor. (See figure 2.0.)

To solve the problem in this case, we have to back up a bit and start with our basic kinematics equations. The horizontal distance traveled by the projectile in time t is

$$x = v_x t \tag{2.1}$$

where v_x is the horizontal component of the initial velocity, $v_x = v_o \cos\theta$. The time t at which the projectile hits the ground (and the target) is found by solving

$$0 = y_o + v_{iy} t - \frac{1}{2} g t^2 \tag{2.2}$$

where $v_{iy} = v_o \sin\theta$ and $g = 9.8$ m/s^2. The solution to equation 2.2 is given by the quadratic equation, so

$$t = \frac{1}{-g}\left[-v_o \sin\theta \pm \sqrt{v_o^2 \sin^2\theta + 2 y_o g}\right]$$

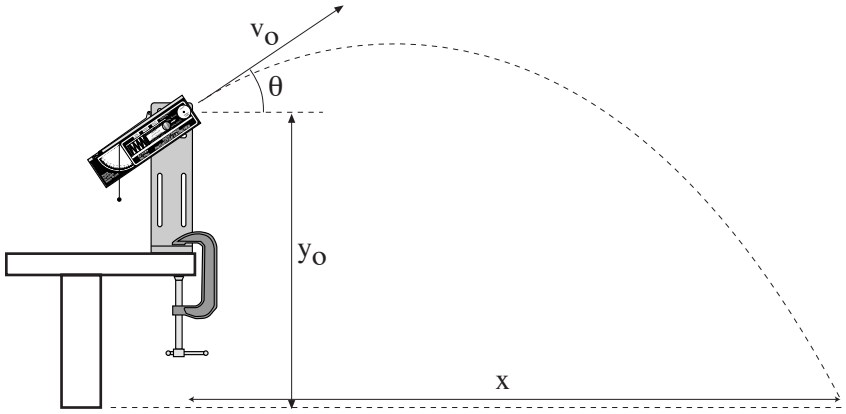


Figure 2.0: Introductory physics laboratory problem

and we want the positive answer, so

$$t = \frac{1}{g}\left[v_o \sin\theta + \sqrt{v_o^2 \sin^2\theta + 2y_o g}\right] \tag{2.3}$$

Plugging equation 2.3 into equation 2.1 gives us our general range equation:

$$x(\theta) = \frac{v_o \cos\theta}{g}\left[v_o \sin\theta + \sqrt{v_o^2 \sin^2\theta + 2y_o g}\right] \tag{2.4}$$

Now we have a problem. It's easy enough to calculate the range x, given θ, but the problem is to find θ to reach a given range x. It is possible to find an algebraic expression for θ from equation 2.4, but it's not easy, or pretty.

One way of solving equation 2.4 is to graph the right-hand side versus t and see where that graph reaches the desired value of x. Or more generally, graph the right-hand side minus the left-hand side, and see where the result crosses the θ axis. (See figure 2.1.) We're going to introduce three methods of solving equations numerically. You may not think of graphing as the most precise method of finding numeric solutions, but it is helpful to keep the graphical model in mind when considering how these numeric methods actually work.

One other factor to keep in mind is that these methods will give you an *approximately* correct answer in all but some special cases. The level of approximation depends on how much computer time you're willing to

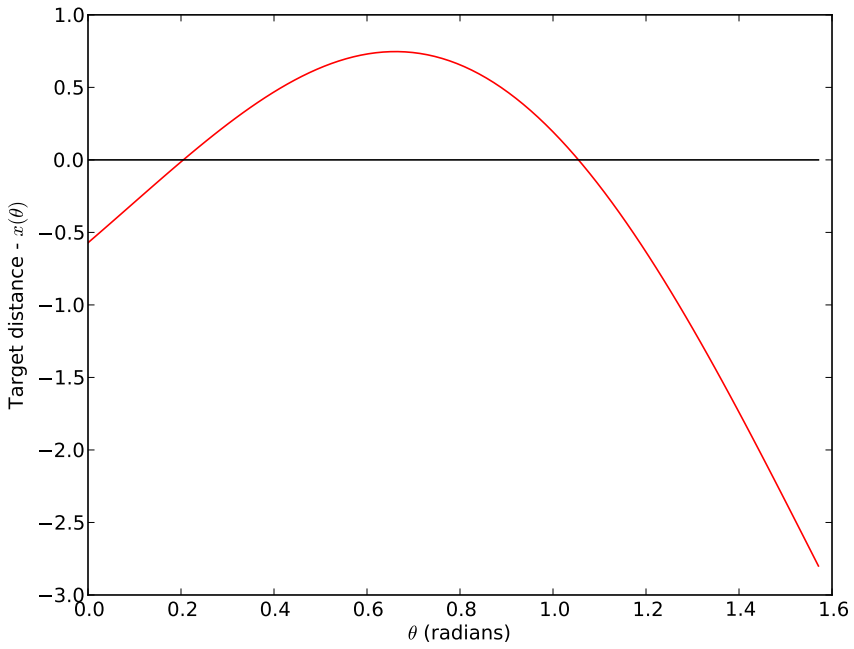


Figure 2.1: Graphical solution for θ, given parameters typical of introductory physics lab.

devote to the problem. You, as the computer user, must decide what level of approximation is good enough.

Bisection Method

To use the bisection method of finding a root, start with two guesses on either side of the root. They don't have to be exact guesses, or even particularly good guesses, but they have to be on either side of the root. We'll call the guess to the left of the root a and the guess to the right of the root b, as shown in figure 2.2.

Next, find the value of the function at the midpoint x between a and b. Compare the signs of $f(x)$ and $f(a)$: if the signs are different, then the root must be between a and x, so let $b = x$. If the signs are the same, then the root must be between x and b, so let $a = x$. Now a and b are such that the solution is still between the two, but they're half as far apart! Repeat this

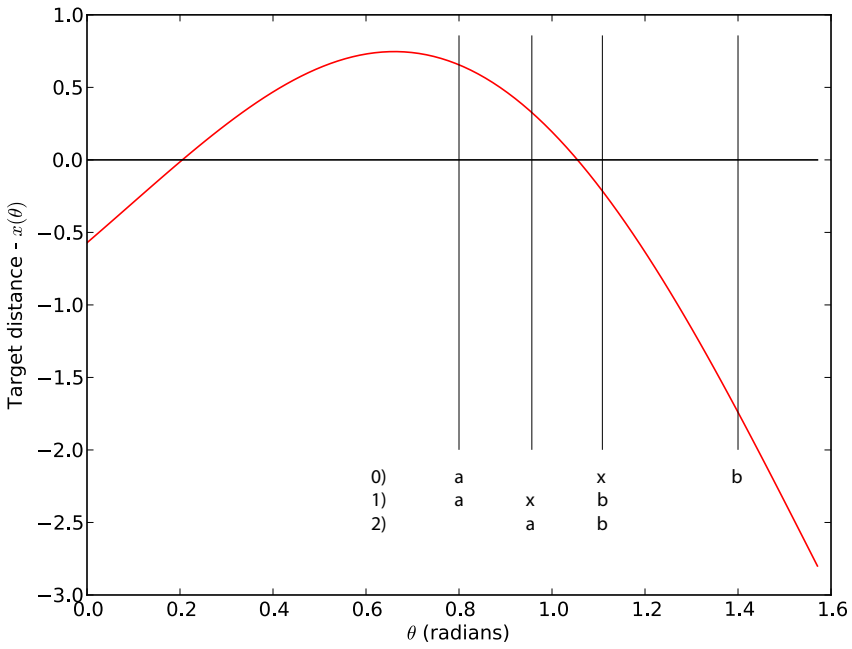


Figure 2.2: Bisection method of finding a root. At step 0, a and b are initial guesses, with a root between them. x is the initial halfway point: since the root is between a and b the new value of b is x. These new values of a and b are used in step 1, and so on.

process until the distance between a and b is less than the desired tolerance for your solution.

Here's a program to solve $f(x) = 0$, in Python code:

Example 2.0.1

```
#!/usr/bin/env python

"""
This program uses the bisection method to find the root
of f(x) = exp(x)*ln(x) -x*x = 0.
"""

from math import *                    # math functions and constants
```

```
tolerance = 1.0e-6          # solution tolerance.

def f(x):                   # function definition: this is the function
                            # to which we are finding a (the) root.
    f = exp(x)*log(x) - x*x
    return f

# Get the initial guesses
a,b = input("Enter two guesses, separated by commas: ")

dx = abs(b-a)               # initial value of dx

while dx > tolerance:       # Repeat until dx < tolerance
    x = (a+b)/2.0
    if (f(a)*f(x)) < 0:     # root is in left half
        b = x
    else:                   # root is in right half
        a = x
    dx = abs(b-a)           # update uncertainty in root location

print 'Found f(x) = 0 at x = %.8f +/- %0.8f ' % (x, tolerance)
```

Note that in the bisection program above, the function could be *any* function that returns a numeric value. In the example it's a relatively simple numeric function, but it doesn't have to be a numeric function in the mathematical sense. You could just as well have a function that — for example — runs a complete simulation and returns the final result. If you can write a function that returns a continuous numeric result, you can use that function with the bisection method.

With this in mind, it would be very handy to have a bisection-method root-finding function that could be used in any general case. Something that you could call like this would be ideal:

```
answer = root_bisection(Equation, FirstGuess, SecondGuess)
```

Here's how to do just that:

Example 2.0.2

```
def root_bisection(f, a, b, tolerance=1.0e-6):
    """
    Uses the bisection method to find a value x between a and b
    for which f(x) = 0, to within the tolerance given.
```

```
    Default tolerance is 1.0e-6, if no tolerance is specified in
    the function call.
    """
    dx = abs(b-a)                   # initial value of dx
    while dx > tolerance:           # Repeat until dx < tolerance
        x = (a+b)/2.0
        if (f(a)*f(x)) < 0:         # root is in left half
            b = x
        else:                       # root is in right half
            a = x
        dx = abs(b-a)               # update uncertainty in root location

    return x                        # answer is x +/- Tolerance
```

You can then include this function in any Python program, and use it to find the root of any function you define. Just pass the *name* of that function to root_bisection().

Example 2.0.3
You want to find the value of θ for which $\cos\theta = 0$.

```
from math import *
from root_bisection import *

theta_0 = root_bisection(cos, 0, pi)
```

The bisection method is simple and very robust. If there is a root between the two initial guesses, bisection *will* find it. Bisection is not particularly fast, though. Each step improves the accuracy only by a factor of two, so if you start with a and b separated by something on the order of 1, and you have a desired tolerance on the order of 10^{-6}, it will take roughly 20 steps. There are faster methods for finding roots.

Newton's Method

Newton's method of rootfinding requires that one know both the function $f(x)$ and its derivative $f'(x) = \frac{df(x)}{dx}$.

Start with a guess a, and calculate the values of $f(a)$ and $f'(a)$. (see figure 2.3.) Use the point-slope form of a line to find the new point b at

which the slope from $f(a)$ intersects the line $y = 0$.

$$f(a) = f'(a)(a - b) + 0 \Longrightarrow b = a - \frac{f(a)}{f'(a)}$$

Repeat the process, starting from point b, and continue repeating until the change from one step to the next is less than the maximum permissible error.

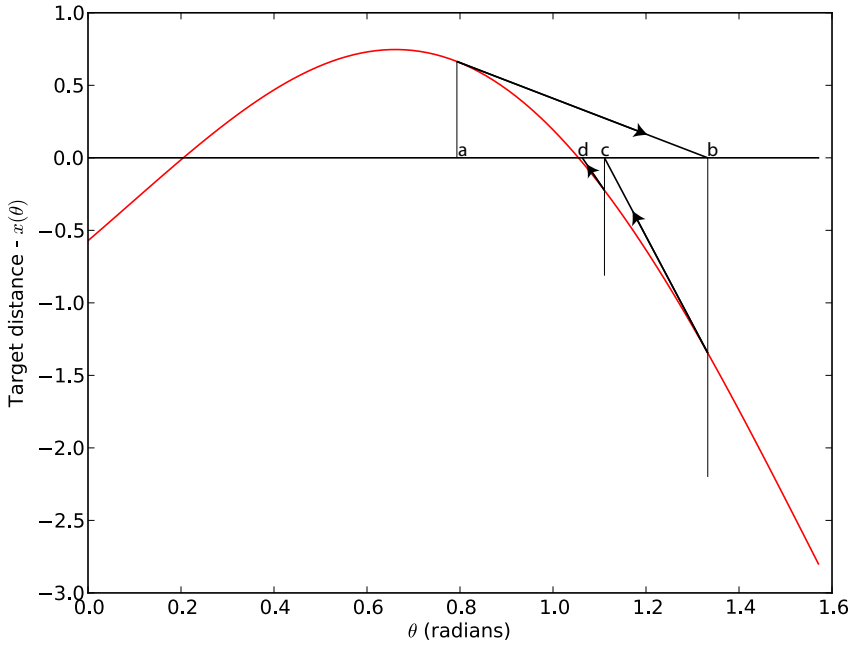


Figure 2.3: Newton's method of finding a root. a is the initial guess. The value of the function at a and the slope of the function at a leads us to the next approximate solution at b. The function value and slope at b lead to the approximate solution at c, and so on until the change between two successive approximations is less than the desired tolerance.

Example 2.0.4

```
def root_newton(f, df, guess, tolerance=1.0e-6):
    """
    Uses Newton's method to find a value x near "guess"
    for which f(x) = 0, to within the tolerance given.
```

```
    Default tolerance is 1.0e-6, if no tolerance is
    specified in the function call.

    It is required to pass this function both f(x) and f'(x).
    """

    dx = 2*tolerance          # initial dx > delta
    while dx > tolerance:     # main loop -- loop until
                              # dx < tolerance
        x1 = x - f(x)/df(x)   # Point-slope form of line
        dx = abs(x-x1)        # how much have things changed?
        x = x1                # Here's the new value

    return x                  # Best value so far
```

Newton's method is generally much faster than the bisection method. It is not as robust, though. For best results, your initial guess should be near the actual solution, and should not be separated from the solution by any local extremes. In addition, Newton's method does not work well near discontinuities in f. The biggest disadvantage to Newton's method, though, is that it requires knowledge of $f'(x)$. If you don't know, or can't obtain, a function that describes the derivative, then Newton's method is not an option.

Secant Method

The secant method is a modification of Newton's method which has the advantage of not needing the derivative function.

Start with *two* guesses, a and b. These should be near the desired solution, as with Newton's method, but they don't have to bracket the solution like they do with the bisection method. Use the values of $f(a)$ and $f(b)$ to approximate the slope of the curve, instead of using a function $f'(x)$ to find the slope exactly, as was done in Newton's method. (see figure 2.4.)

Find the value of x at which the line through points $(a, f(a))$ and $(b, f(b))$ hits the x axis. Next, repeat the process with the new point and the initial guess that is closest to the new point. Keep repeating until the change from one iteration to the next is less than the desired tolerance.

Like Newton's method, the secant method is not as robust as the bisection method. It can be unpredictable when used on discontinuous functions, and the initial guesses should be relatively close to the desired root. It is nearly as fast as Newton's method, though, and it has the advantage of not

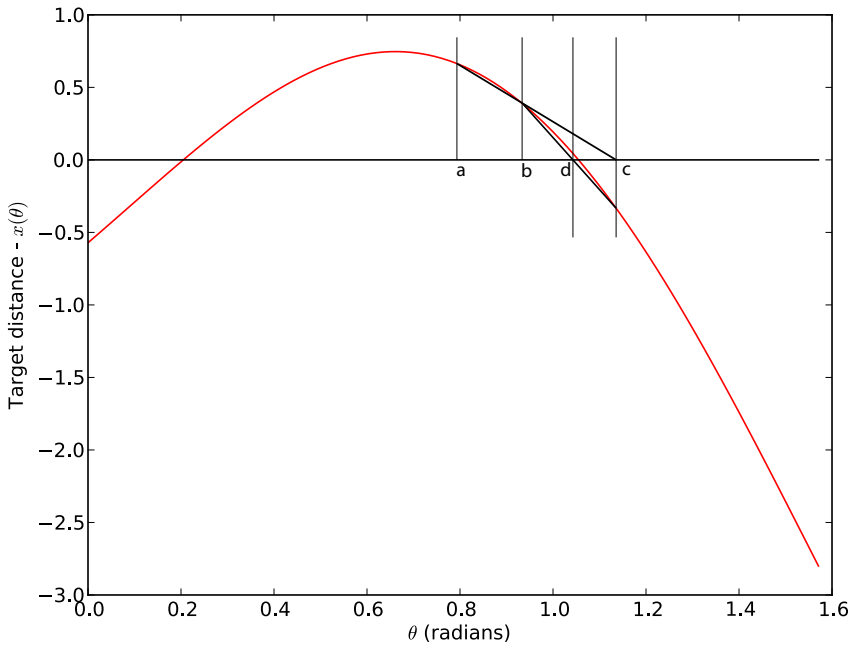


Figure 2.4: Secant method of finding a root. a and b are the initial guesses. The line through $(a, f(a))$ and $(b, f(b))$ leads us to the next approximate solution at c. The process is repeated until the change between two successive approximations is less than the desired tolerance.

needing a derivative. The secant method is a good choice for non-oscillatory smooth functions, such as usually appear in physics problems.

2.0.1 Python Libraries

Finding roots is a common-enough problem that people have written some good libraries to take care of it for you. One of those libraries is scipy.optimize, a subset of the "scientific Python" (scipy) library. Scipy includes implementations of the bisection method (`bisect()`), Newton's method (`newton()`), and several others including the `brentq()` routine. For a complete description of the Brent algorithm, see chapter 9 of [13]: the short version is that by combining root-bracketing, bisection, and inverse quadratic interpolation the routine can find roots about as fast as the secant method while *guaranteeing*

to find an existing solution between the given endpoints. Details on how to use brentq() are given in the scipy reference guide on the web: an example follows.

Example 2.0.5

```
from pylab import *
from scipy.optimize import brentq
x = brentq(sin, 2, 4)    # will find value of x between 2 and 4
                         # for which sin(x) = 0
print x, x-pi
```

$$\Longrightarrow 3.141592653589793, 0.0$$

It's important to know how to "roll your own"; to be able to write routines for basic tasks such as root-finding. It is also important to know when to use existing code. The brentq() routine is a good example of when to use existing code. Unless you are finding roots of a pathologically difficult function, brentq() will find the solution quickly and accurately and save you a lot of work.

2.1 Numeric Integration

Integration is an area in which numerical methods can be of considerable help. Take the function shown in figure 2.5. Depending on the functional form of $f(x)$, it may not be possible to calculate the integral of f from a to b analytically.

The integral of $f(x)$ from a to b is the area under the curve, though, which leads to the earliest numeric method of estimating that integral: carefully graph the function onto cardboard of known density and cut the shaded area out. From the weight of the cut section and the density of the cardboard, one can calculate the area and thus the value of the integral.

There are, of course, more precise methods of estimating integrals. Bear in mind, though, that all of the methods given here are methods of *estimation*. Do not blindly assume that because you use a given method that your answer is correct.

Simple Method

The simplest method of estimating the integral shown in figure 2.5 is to divide the integration range into N slices of width dx. Calculate the value

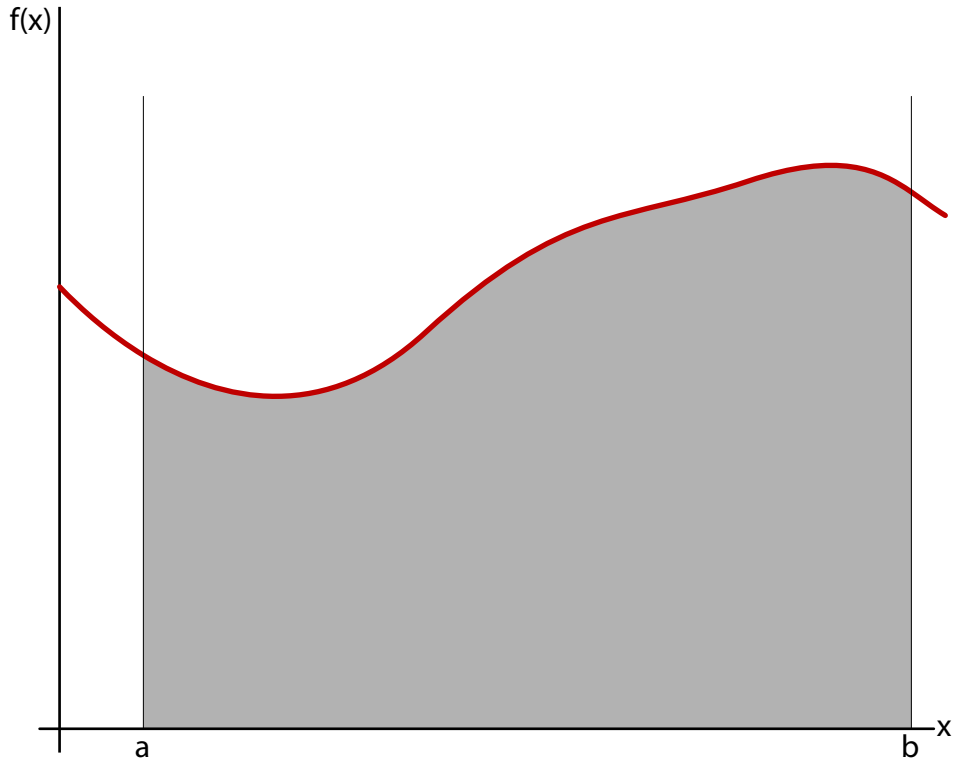


Figure 2.5: Some function $f(x)$, integrated from a to b. The exact value of the integral is the shaded region.

of $f(x_i)$ at some point on each slice, and find the area $A_i = f(x_i)\Delta x$ for each slice, as shown in figure 2.6. The integral is then approximately

$$\int_a^b f(x)\,dx \approx \sum_i f(x_i)\Delta x$$

Example 2.1.1
A python implimentation of the simple integration method could look something like this:

```
def int_simple(fn, a, b, N):
    """
    A routine to do a simple a rectangular-slice approximation
    of an integral.
```

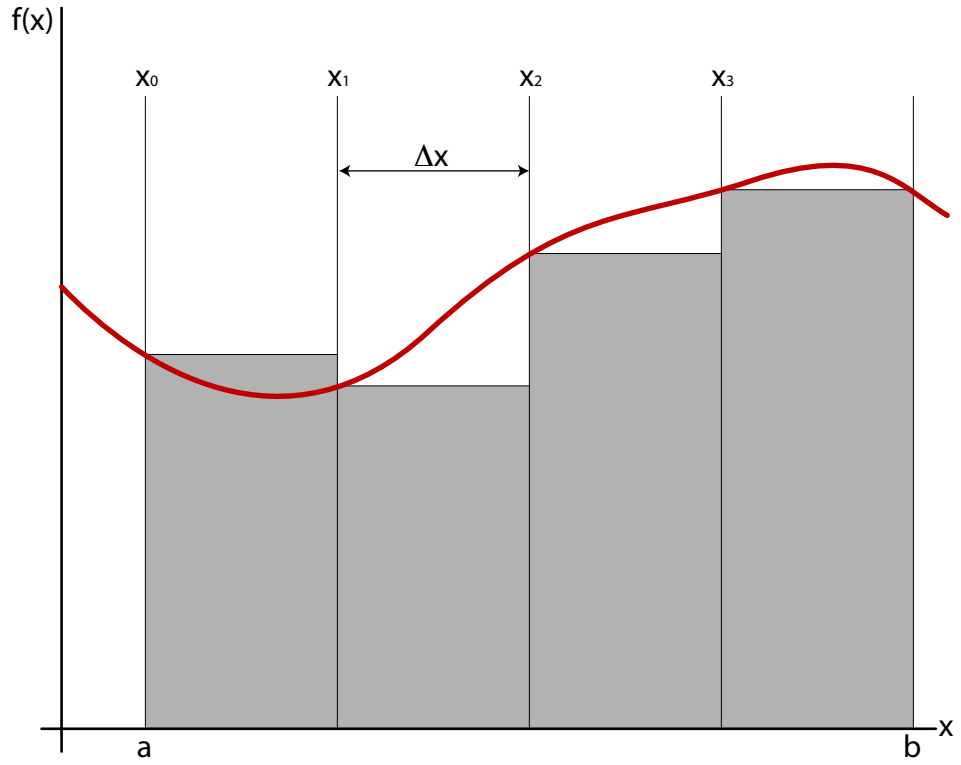


Figure 2.6: Simplest method of numerical integration. Here the value of $f(x)$ is calculated at the left edge of each slice: the final point $f(b)$ is not used in this calculation.

```
    fn: the function to integrate
    a,b: limits of integration
    N: number of slices to take
"""
I = 0                   # The value of the integral
dx = (b-a)/float(N)     # the stepsize, or width of slice
for j in range(N):
    x = a + dx * j      # The x value for this slice
    I = I + fn(x)*dx    # add the area of this slice
return I
```

This method is very simple to use and to understand, but as you can see from figure 2.6 it's not particularly good. If you increase the number of

slices, the estimate becomes better: in fact the exact value of the integral is

$$\int f(x)\,dx = \lim_{\Delta x \to 0} \sum_i f(x_i)\Delta x$$

so if you use a slice width of zero, the answer is exact. The problem, of course, is summing the infinite number of slices that accompany a slice width of zero!

There is a limited return on investment with an increasing number of slices. The error in each slice is the approximately triangular piece at the top. If you use twice as many slices, each of half the width, of the original slice, then that triangle becomes two triangles, each one roughly half the linear size of the original. The area of the smaller triangles is each 1/4 the area of the original, but there are two of them, so the area that is missed is cut by half overall.

There are more precise methods of doing that rough error calculation, but the results are the same: The precision of the result for the simple method goes as N. Doubling the number of slices takes you twice as long, since you must calculate $f(x)$ for twice as many points, and the uncertainty in your result is half what it was.

There are better ways of approximating the integral, but before we go on to them let's take a moment or two to improve the way we call an integrating function. In example 2.1.1, there were actually two different things that took place: calculation of the function to obtain the function values at regular intervals, and summing of those values to calculate the approximate value of the integral. Although these two seem to be parts of the same process, there are two significant advantages to treating them separately. First, not all integrations are integrations of functions. It is often necessary to integrate data, in which case there is no function to call to find the next value of $f(x)$. Instead, the most likely form of $f(x)$ is that of a list of data values. Second, the simple integration method shown in example 2.1.1 is somewhat unusual in that it uses the value of the function at any point only once. The more precise methods described in the next few pages use the function values more than once.

Because of these two reasons, it's best to design our integration routines to accept a list of function values and the spacing Δx between those values. This way the routine can be used more generally. For data sets, the calling program passes the list of values to the integration routine. For functions, the calling program calculates a list of function values once, then passes that list to the same integration routine.

Example 2.1.2

```
def int_simple(f,dx):
    """
    Simplest integration possible, using uniform rectangular
    slices. Gives a rather poor result, with error on the
    order of dx.

    f[] should be a list of function values at x values
    separated by the interval dx. The limits of integration
    are x[0] -> x[0]+dx*len(f).

    Note that this algorithm does not use the last point in f!
    """
    return dx*sum(f[0:-1])  # The sum() function is built-in,
                            # and does # exactly what you would
                            # expect.

# What follows is a program to use the int_simple() function
# to calculate the integral of sin(x) from 0 to pi.

from math import *

N = 100                     # number of slices.
                            # Higher N gives better results.
a = 0.0
b = pi
interval = (b-a)/float(N)

# Calculate the values of x to use.
x = [a + interval * i for i in range(N+1)]
# notice that it takes N+1 function values to define N slices!

# Calculate the values of f(x).
FunctionValues = [sin(value) for value in x]

print "The value of the integral is approximately",
print int_simple(FunctionValues, interval)
```

Now we're ready to move on to better integration methods.

Trapezoid Method

We can greatly improve the efficiency of our integration by approximating the slices as trapezoids instead of as rectangles, as shown in figure 2.7.

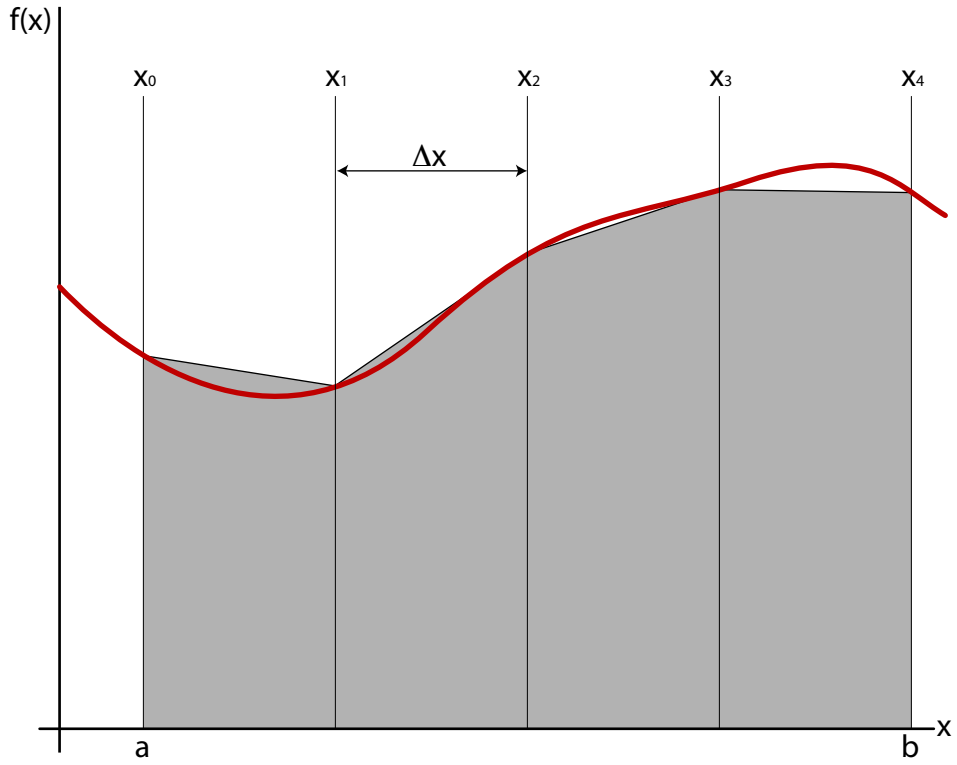


Figure 2.7: The trapezoid method of numerical integration. The value of $f(x)$ at both sides of each slice is used to calculate the area of each trapezoid.

The area each trapezoid is the width Δx times the average height of the slice sides:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} \frac{f(x_i) + f(x_{i-1})}{2} \Delta x \tag{2.5}$$

If you look at the right hand side of equation 2.5 closely, you can convince yourself that it works out to

$$\int_a^b f(x)\,dx \approx \left[\frac{f(x_0) + f(x_N)}{2} + \sum_{i=1}^{N-1} f(x_i) \right] \Delta x \tag{2.6}$$

This is probably the most efficient way to code the trapezoid method, although the actual function is left as an exercise.

Something to note about terminology, here: there's a difference between *points* and *slices*. In figure 2.7, there are four slices (the four grey trapezoids.) It takes five points (x_0–x_4) to define those four slices. Make sure in your code that it's clear whether N is the number of points or the number of slices, because if you're not careful either your value of dx or your range of integration will be incorrect! For large N, it makes a very small error, but the point is to avoid using large N.

Simpson's Method

The simple method approximates the function as a constant for the slices. The trapezoid method approximates the function as a linear equation for each slice. The next level of approximation, called Simpson's method after its inventor, is to approximate the function as a collection of parabolas, with each pair of slices providing the three points necessary to define the parabola for that region. (See figure 2.8.) The value of the integral is then approximately

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{\frac{N-1}{2}} \int_{x_{2i-1}}^{x_{2i+1}} g_i(x)\,dx \tag{2.7}$$

where the $g_i(x)$ are the parabolas through the sets of three points described previously.

The derivation of Simpson's method is somewhat complicated, but worth considering carefully. Start with the equation for a parabola which goes through the three points $(x_{k-1}, f(x_{k-1}))$, $(x_k, f(x_k))$, and $(x_{k+1}, f(x_{k+1}))$:

$$\begin{aligned} g_k(x) =& \frac{(x-x_k)(x-x_{k+1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})} f(k-1) \\ &+ \frac{(x-x_{k-1})(x-x_{k+1})}{(x_k-x_{k-1})(x_k-x_{k+1})} f(k) \\ &+ \frac{(x-x_{k-1})(x-x_k)}{(x_{k+1}-x_{k-1})(x_{k+1}-x_k)} f(k+1) \end{aligned} \tag{2.8}$$

(Remember that $g_k(x)$ is the parabolic approximation of a section of the function $f(x)$, and $f(x)$ is what we're actually integrating.) We re-order

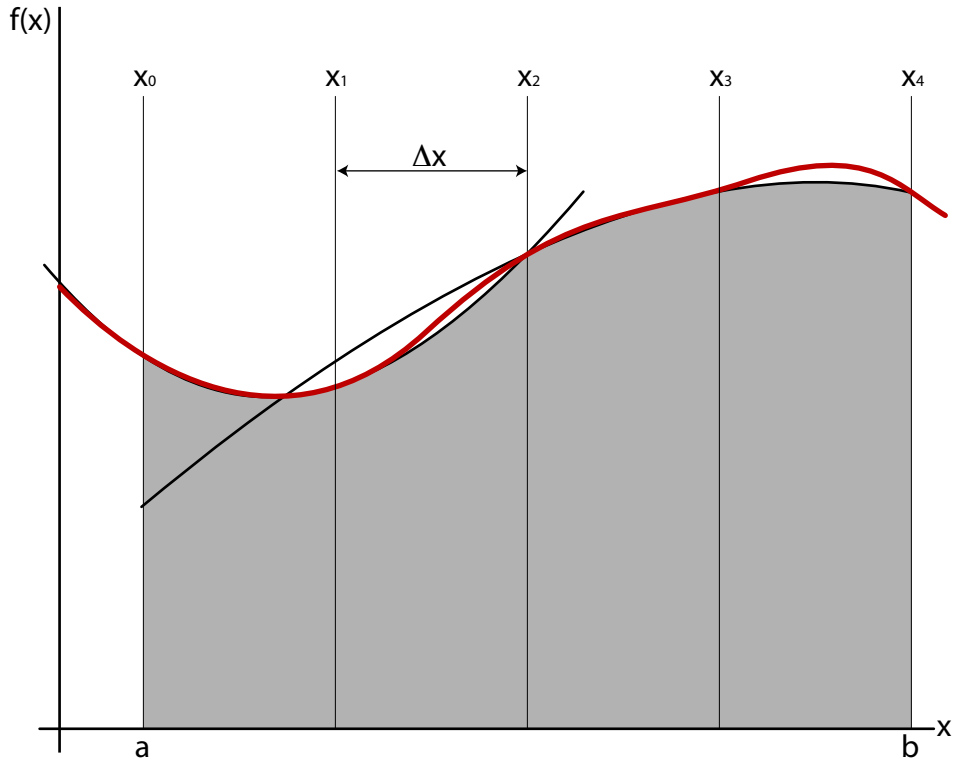


Figure 2.8: Simpson's method of integration. Each pair of slices is used to form a parabola which goes through the three function values associated with that pair. This example shows four slices, with two approximation parabolas.

equation 2.8 in terms of powers of the continuous variable x:

$$
\begin{aligned}
g_k(x) = x^2 & \left[\frac{f(x_{k-1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})} + \frac{f(x_k)}{(x_k-x_{k-1})(x_k-x_{k+1})} + \frac{f(x_{k+1})}{(x_{k+1}-x_{k-1})(x_{k+11}-x_k)} \right] \\
& + x \left[\frac{-(x_k+x_{k+1})f(x_{k-1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})} + \frac{-(x_{k-1}+x_{k+1})f(x_k)}{(x_k-x_{k-1})(x_k-x_{k+1})} + \frac{-(x_{k-1}+x_k)f(x_{k+1})}{(x_{k+1}-x_{k-1})(x_{k+11}-x_k)} \right] \\
& + \left[\frac{x_k x_{k+1} f(x_{k-1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})} + \frac{x_{k-1}x_{k+1}f(x_k)}{(x_k-x_{k-1})(x_k-x_{k+1})} + \frac{x_{k-1}x_k f(x_{k+1})}{(x_{k+1}-x_{k-1})(x_{k+11}-x_k)} \right] \\
& \equiv x^2[A] + x[B] + [C] \qquad (2.9)
\end{aligned}
$$

Each of the terms $[A]$, $[B]$, and $[C]$ depend only on the numeric values at the edges of the slices for the integrations.

The integral of $g_k(x)$ is just

$$\int_{k-1}^{k+1} g_k(x)\,dx = \frac{1}{3}[A](x_{k+1}^3 - x_{k-1}^3) + \frac{1}{2}[B](x_{k+1}^2 - x_{k-1}^2) + [C](x_{k+1} - x_{k-1}) \tag{2.10}$$

After a few pages of careful algebra, the combination of equations 2.7–2.10 gives us a final result:

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3} \sum_{k=odd}^{N-1} (f(x_{k-1}) + 4f(x_k) + f(x_{k+1})) \tag{2.11}$$

The algorithm described above requires that there be an even number of slices. If there is an odd number of slices, it's necessary to add the area of the last slice separately. The derivation of that area is left as an exercise to the student, and the result is

$$A_{\text{last slice}} = \frac{\Delta x}{12}[5f(N) + 8f(N-1) - f(N-2)] \tag{2.12}$$

A Simpson's method integration function is shown in example 2.1.3.

Example 2.1.3

```
def int_simpson(f,dx):
    """
    Simpson's rule, using uniform slices.

    f[] should be a list of function values, separated on the
    x axis by the interval dx. The limits of integration are
    x[0] -> x[0]+dx*len(f).

    This particular algorithm does not require that there be
    an even number of intervals (odd number of points): instead,
    it adds the last section separately if necessary.
    """

    # number of points
    N = len(f)
    # initial value of integral
    integral = 0.0

    # add up terms
```

```
    for i in range(1, N-1, 2):
        integral = integral + f[i-1] + 4.0*f[i] + f[i+1]
    # multiply by dx, and divide by 3
    integral = integral * dx / 3.0

    # if number of points is even (odd number of slices) then
    # add the last point separately.
    if (N % 2) == 0:
        integral = integral + dx * (5.0*f[-1] + 8.0*f[-2]
                   - f[-3])/12.0

    return integral
```

Other Methods

The error of the simple measure goes as Δx, as we showed previously, so doubling the number of slices decreases the error in the result at a cost of twice the computation time. The error of the trapezoid method goes as Δx^2, so doubling the number of slices decreases the error by a factor of four but only costs twice as much in terms of computation time. That's a nice gain, but Simpson's method is even better. The error in Simpson's method goes as Δx^4 [11], so doubling the number of slices decreases the error by a factor of sixteen. Doubling the slices doubles the time, of course, but Simpson's method is considerably more complicated than the trapezoid or simple method so there's an additional time cost to Simpson's method compared to others.

Intuition might tell you that next method in the series would approximate the integral as a cubic over each set of three slices, and that this method would give you an error that went as some even higher power of Δx. While in theory that intuitive answer is correct, in practice it's not practical to implement the method. The amount of calculation for this third-order method is so high that you can get better results for less work by increasing the number of points on Simpson's method rather than using a cubic method.

There are methods of numeric integration that are specialized for certain classes of functions, and if you are curious about these I encourage you to read more about Gaussian Quadrature (and other methods) in *Numerical Methods in C* [13] or some similarly advanced text. For general use, though, Simpson's method is usually a good place to start.

2.2 Differentiation

One of the tools commonly used in an introductory physics laboratory is the "sonic ranger". This uses pulses of sound to determine the distance between a sensor and some object, typically a dynamics cart. The sonic ranger measures distance only, but we would like to know velocity and acceleration. This is an example where numeric differentiation becomes useful.

One way of finding a numeric derivative is to start from the definition of a derivative:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + x_o) - f(x)}{\Delta x}$$

So we could approximate the derivative by taking the difference between two successive data points and dividing by the distance between them:

$$f'_i \approx \frac{f_{i+1} - f_i}{\Delta x} \tag{2.13}$$

where the notation $f_i \equiv f(x_i)$ is used to indicate the discrete nature of our data and simplify the expression. This simple approach works, but there are better methods.

To find some of these better methods, let's start with the Taylor expansion:

$$f(x) = f(x_o) + (x - x_o)f'(x) + \frac{(x - x_o)^2}{2!}f''(x) + \frac{(x - x_o)^3}{3!}f^{(3)}(x) + \cdots$$

or in discrete notation,

$$\begin{aligned} f_{i+1} &= f_i + (x_{i+1} - x_i)f'_i + \frac{(x_{i+1} - x_i)^2}{2!}f''_i + \frac{(x_{i+1} - x_i)^3}{3!}f^{(3)}_i + \cdots \\ &= f_i + \Delta x f'_i + \frac{\Delta x^2}{2!}f''_i + \frac{\Delta x^3}{3!}f^{(3)}_i + \cdots \end{aligned} \tag{2.14}$$

We can immediately see the simplest method of finding a first derivative from this equation: it comes directly from the first two terms on the right hand side of equation 2.14. We can also see that since the error in this approximation goes as the power of Δx in the last term we actually use, the error in this simplest approximation is Δx.

We could also use the Taylor expansion to approximate the function at f_{i-1}.

$$f_{i-1} = f_i - \Delta x f'_i + \frac{\Delta x^2}{2!}f''_i - \frac{\Delta x^3}{3!}f^{(3)}_i + \cdots \tag{2.15}$$

Subtracting equation 2.15 from 2.14 gives us

$$f_{i+1} - f_{i-1} = 2\Delta x f_i' + 2\frac{\Delta x^3}{6} f_i^{(3)} + \cdots \tag{2.16}$$

If we take the Δx^3 and smaller terms as negligible and solve for f_i', we obtain the "three-point derivative" approximation:

$$f_i' \approx \frac{f_{i+1} - f_{i-1}}{2\Delta x} \tag{2.17}$$

The error in equation 2.17 goes as the power of Δx in the last term we didn't just ignore, which is Δx^2. Computationally, this gives us a much better result than the simplest method for very little extra computational cost.

We can obtain higher precision by taking the expansion of $f_{i\pm 2}$:

$$f_{i+2} - f_{i-2} = 4\Delta x f_i' + 2\frac{8\Delta x^3}{6} f_i^{(3)} + \cdots \tag{2.18}$$

Now multiply equation 2.16 by 8 and subtract equation 2.18 to obtain the "five-point derivative" formula:

$$f_i' = \frac{1}{12\Delta x}\left(f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}\right) \tag{2.19}$$

The error in this "five-point derivative" formula goes as Δx^4, since the 4^{th}-order term in the Taylor expansion cancels in each of equations 2.16 and 2.18, so our first neglected term is 5^{th} order.

We can continue the process to obtain higher-order approximations, at the cost of increased complexity. A disadvantage of higher-order approximations, in addition to complexity, is that derivatives near the edges of the initial data set are lost. The five-point derivative formula can only calculate derivatives at points 3 through $N - 2$.

Higher-order derivatives are available by the same process. For example, adding equations 2.14 and 2.15 gives us

$$f_{i+1} + f_{i-1} = 2f_i + \Delta x^2 f_i'' + \cdots \tag{2.20}$$

so

$$f_i'' \approx \frac{f_{i-1} - 2f_i + f_{i+1}}{\Delta x^2} \tag{2.21}$$

where the error is on the order of Δx^3, since the third-derivative terms cancel perfectly in equation 2.20. This is the "three-point second-derivative

formula". The "Five-point second-derivative formula" is given —through a similar process— by

$$f_i'' \approx \frac{1}{12\Delta x^2}\left(-f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+2}\right) \tag{2.22}$$

As you might expect, the scipy library has a derivative function also.[1] It's scipy.misc.derivative(), and it uses a central difference formula (as described above) to calculate the n^{th} derivative at a given point. It expects a function, though, rather than a list of data points. It should be called with the function name, the point at which one wants the derivative, and (optionally) the value of dx, the order n of the derivative, optional arguments to the function, and the number of points to use.

```
from scipy.misc import derivative
print derivative(sin, pi)
>>> -0.8414709848078965
```

You will notice that the value returned by the above code is not -1. By default, $dx = 1.0$ and the order is the three-point approximation. To get better results, use a smaller value of dx and at least a 5-point approximation:

```
print derivative(sin, pi, dx=0.1, order=5)
>>> -0.99999667063260755
```

[1] `http://docs.scipy.org/doc/scipy/reference/generated/scipy.misc.derivative.html`

2.3 Problems

2-0 Write a generalized function implementing the secant method of root-finding, similar to example 2.0.2.

2-1 Write a program that uses the trapezoid method to return the integral of a function over a given range, using a given number of sample points. The actual calculation should be a function of the form int_trap(f,dx), where f is a list of function values and dx is the slice width.

2-2 Compare the results of the simple integration method, the trapezoid integration method from problem 1, and Simpson's method of integration for the following integrals:

(a)
$$\int_0^{\pi/2} \cos x \, dx$$

(b)
$$\int_1^3 \frac{1}{x^2} \, dx$$

(c)
$$\int_2^4 x^2 + x + 1 \, dx$$

(d)
$$\int_0^{6.9} \cos\left(\frac{\pi}{2}x^2\right) \, dx$$

For each part, try it with more and with fewer slices to determine how many slices are required to give an 'acceptable' answer. (If you double the number of slices and still get the same answer, then try half as many, etc.) Parts (c) and (d) are particularly interesting in this regard. In your submitted work, describe roughly how many points were required, and explain.

Note: The function in (d) is the *Fresnel Cosine Integral*, used in optics. It may be helpful in understanding what's going on with your integration if you make a graph of the function. For more information on this function, see [13].

2-3 Show that equation 2.12 is correct.

2-4 Write a program that can calculate double integrals. Use the simple method, but instead of using rectangular slices use square prisms with volume $f(x, y)\Delta x\Delta y$ Check your program by using it to calculate the volume of a hemisphere.

2-5 The Simpson's method program developed in this chapter requires uniform slice width. It's sometimes convenient to integrate using slices of varying width, for example if one needs to integrate data sets that are taken at irregular time intervals. Write a Simpson's method routine that integrates over non-constant-width slices. The routine should take two arrays as its arguments: the first array should be an array of function values $f(x)$, and the second an array of values of x.

2-6 The Fermi-Dirac distribution describes the probability of finding a quantum particle with half-integer spin ($\frac{1}{2}, \frac{3}{2}, \cdots$) in energy state E:

$$f_{FD} = \frac{1}{e^{(E-\mu)/kT} + 1}$$

The μ in the Fermi-Dirac distribution is called the *Fermi energy*, and in this case we want to adjust μ so that the total probability of finding the particle *somewhere* is exactly one.

$$\int_{E_{min}}^{E_{max}} f_{FD}\, dE = 1$$

Imagine a room-temperature quantum system where for some reason the energy E is constrained to be between 0 and 2 eV. What is μ in this case? At room temperature, $kT \approx \frac{1}{40}$ eV. Feel free to use any of the integration and/or root-finding routines you've learned in this chapter.

2-7 Write a function that, given a list of function values f_i and the spacing dx between them, returns a list of values of the first derivative of the function. Test your function by giving it a list of known function values for $\sin(x)$ and making a graph of the differences between the output of the function and $\cos(x)$.

2-8 Write a function that, given a list of function values f_i and the spacing dx between them, returns a list of values of the second derivative of the function. Test your function by giving it a list of known function values for $\sin(x)$ and making a graph of the differences between the output of the function and $-\sin(x)$.

2-9 Derive equation 2.22.

Chapter 3

Numpy, Scipy, and MatPlotLib

3.0 Numpy

Python lists are quite powerful, but they have limitations. Take matrices, for example: a two-dimensional array can be expressed as a list of lists.

```
M = [ [1,2], [3,4] ]
```

The result is something that is very much like the matrix

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} .$$

Addressing elements in this "matrix" is clumsy: the top-right element would be addressed as M[0][1] instead of the more intuitive M[0,1]. More bothersome to us in a computational physics course, though, is that although a list of lists looks something like a matrix it doesn't behave like a matrix mathematically!

```
print M*3
[[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]]
print M+M
[[1, 2], [3, 4], [1, 2], [3, 4]]
print M*M
TypeError: can't multiply sequence by non-int of type 'list'
```

There is, of course, a package to provide real matrices to Python. Numpy, short for "Numerical Python", provides everything we need and much more, in the form of numpy "arrays". Setting up an array is similar to what you'd do for a list of lists, but with the specification that the item is an array:

```
from numpy import *
M = array([ [1,2], [3,4] ])
print M
[[1  2]
 [3  4]]
```

Note that the array is automatically printed in an array format. Scalar multiplication now works as expected:

```
print M*3
[[ 3  6]
 [ 9 12]]
```

As does addition:

```
print M+M
[[2  4]
 [6  8]]
```

There are at least three different ways of multiplying matrices: the default is scalar multiplication of matrix elements:

```
print M*M
[[ 1  4]
 [ 9 16]]
```

But of course one can also do dot and cross product multiplications with numpy:

```
print dot(M,M)
[[ 7 10]
 [15 22]]
print cross(M,M)
[0 0]
```

In fact, most common matrix operations are built in to numpy:

```
print transpose(M)
[[1  3]
 [2  4]]
print inv(M)
[[-2.   1. ]
 [ 1.5 -0.5]]
print det(M)
-2.0
print eig(M)
(array([-0.37228132,  5.37228132]), array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]]))
```

Numpy uses the LINPACK linear algebra package, which is written in FORTRAN and is extremely fast. The fact that arrays are multiplied element-wise by default is a great advantage. It means that if you have large lists of numbers that need to be multiplied together, you can define each array of numbers as a numpy array and then just mulitply the arrays. Doing the multiplications this way is much faster than using a **for** loop, and generally easier as well. Similarly, functions that would normally calculate single values will usually calculate array values if given an array input.

Example 3.0.1
Calculate sine of $\{\pi/6, \pi/5, \pi/4, \pi/3, \pi/2\}$.

```
x = array([pi/6, pi/5, pi/4, pi/3, pi/2])
print sin(x)
[ 0.5        0.58778525  0.70710678  0.8660254   1.
]
```

The numpy package also includes *many* other functions of interest to physicists and mathematicians.

Example 3.0.2
You are given a system of linear equations as follows, and need to find the values of w, x, y, and z:

$$\begin{aligned} w + 3x - 5y + 2z &= 0 \\ 4x - 2y + z &= 6 \\ 2w - x + 3y - z &= 5 \\ w + x + y + z &= 10 \end{aligned}$$

The solution, using numpy, is to rewrite the system of equations as a matrix multiplication:

$$\begin{bmatrix} 1 & 3 & -5 & 2 \\ 0 & 4 & -2 & 1 \\ 2 & -1 & 3 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 6 \\ 5 \\ 10 \end{bmatrix}$$

or

$$Mx = b$$

So we define M and b and solve using the linalg.solve() function of numpy:

```
M = array([ [1,3,-5,2], [0,4,-2,1], [2,-1,3,-1], [1,1,1,1] ])
b = array([0,6,5,10])
x = linalg.solve(M,b)
print x
[ 1.  2.  3.  4.]
```

We will be using many numpy functions throughout the rest of the course, but there are several other tools in numpy that we use frequently enough to mention specifically here.

arange() Creates an "Array Range". This is just like the range() function built into python, except it returns a numpy array and *the step doesn't have to be an integer.*

```
print arange(0,1, 0.25)
[ 0.    0.25  0.5   0.75]
print arange(0,1.01, 0.25)
[ 0.    0.25  0.5   0.75  1.  ]
```

linspace() Creates a "linearly-spaced" array. This is similar to arange() except instead of the step value you tell it the number of points you want. The resulting array will contain both endpoints and evenly-spaced values between those points.

```
print linspace(0,1,5)
[ 0.    0.25  0.5   0.75  1.  ]
```

logspace() Just like linspace() but the values are spaced so that they are evenly distributed on a logarithmic scale. The stop and start values should be given as the powers of 10 that are desired.

```
print logspace(1, 3, 4)
[    1.    10.   100.  1000.]
```

zeros() Produces a numpy array filled with zeros of the specified type. This is useful for setting up an empty array that your program can then fill with calculated values. The "type" can be float or int or double (or others) and specifies what format should be used for storing values. The default type is float.

```
print zeros([2,3], int)
[[0 0 0]
 [0 0 0]]
```

ones() Just like zeros() but fills in ones instead.

Numpy arrays are the standard format for any numeric work in Python, and are the expected format for Scipy and MatPlotLib.

3.1 Scipy

As one might guess, "Scipy" is short for "Scientific Python" and it is a pack of extensions to Python that provides numerous scientific tools. Where numpy provides basic numeric tools such as the ability to do nice things with matrices, Scipy provides higher-level tools such as numeric integration, differential equation solvers, and so on.

In the previous chaptor we derived a method for calculating integrals using Simpson's method: Simpson's method is included in Scipy.

```
from scipy.integrate import simps
```

The documentation for this function shows that it has some extra features not discussed previously: pydoc scipy.integrate.simps to see these features.

Scipy also provides the ability to integrate functions (rather than lists of y values) using Gaussian Quadrature.

```
from scipy.integrate import quad
print quad(sin, 0, pi)
(2.0, 2.2204460492503131e-14)
```

The quad() function takes any function (sine, in this case) and integrates over the range given (0, pi). It returns a tuple containing the value of the integral (2.0) and the uncertainty in that value.

If you need to integrate to infinity, scipy provides for that also:

```
from scipy.integrate import inf
print quad(lambda x: exp(-x), 0, inf)
(1.0000000000000002, 5.8426067429060041e-11)
```

Scipy also provides Fourier transforms, differential equation solvers, and other useful tools that will be described later in this text.

3.2 MatPlotLib

"A picture is worth a thousand words," as the saying goes, and nowhere is that more true than in this field. For all but the most simple numeric problems, graphs are the best way of showing your results. The MatPlotLib

library provides a powerful and easy set of tools for two-dimensional graphing.

Example 3.2.1
Plot sine and cosine over the range $\{-\pi, \pi\}$.

```
from pylab import *
x = arange(-pi, pi, pi/100)
plot(x, sin(x), 'b-', label='sine')
plot(x, cos(x), 'g--', label='cosine')
xlabel('x value')
ylabel('trig function value')
xlim(-pi, pi)
ylim(-1,1)
legend(loc='upper left')
show()
```

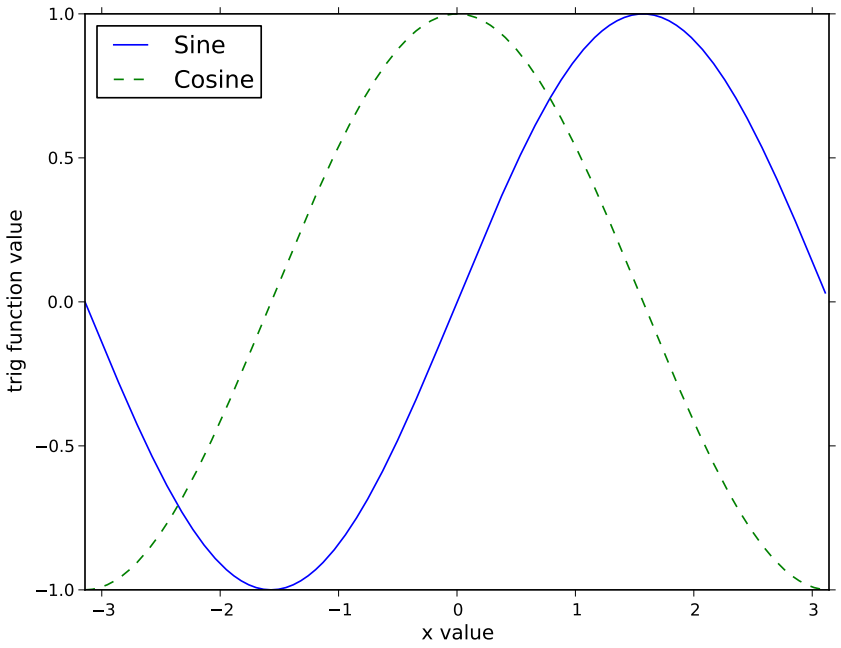


Figure 3.0: Figure produced in example 3.2.1.

Character	Line Type
'-'	solid line
'–'	dashed line
':'	dotted line
'-.'	dash-dot line

Table 3.1: MatPlotLib line types

Character	Datapoint indicator
'o'	circle
'^'	upward-pointing triangle
's'	square
'+'	plus
'x'	cross
'D'	diamond

Table 3.2: MatPlotLib datapoint types

Let's go over the details of example 3.2.1.

```
from pylab import *
```

The pylab package is a wrapper package that imports numpy, scipy, and matplotlib. It indirectly takes care of trig functions and π also, which makes it a nice start to everything we're doing.

```
x = arange(-pi, pi, pi/100)
```

This sets up an array of x values, 100 of them, from $-\pi$ to π.

```
plot(x, sin(x), 'b-', label='sine')
```

This sets up the plot of sin(x). The first parameter is the array of x-axis values, the second is the array of y-axis values. (Remember that numpy arrays can be calculated "all at once": since x is an array, $\sin(x)$ is an array containing $\sin(x_i)$ for each value x_i in the array x.)

The third item in the plot function call specifies the line style, in this case a blue (b) solid line (-).

The various characters in tables 3.1–3.3 can be combined, so for a green dotted line connecting circles at the datapoints one would specify 'go:'.

The label keyword sets what is to be displayed in the legend. The xlabel() and ylabel() set the labels on the x and y axes, respectively. The xlim()

Character	Color
'b'	blue
'c'	cyan
'g'	green
'k'	black
'm'	magenta
'r'	red
'w'	white
'y'	yellow

Table 3.3: MatPlotLib built-in colors

and ylim() function calls set the range displayed on the graphs. These are optional, and matplotlib is pretty good at selecting reasonable values if you don't specify any. The legend() call sets various characteristics of the legend — in this case it is used to set the location.

The final function call in the example is show(), which draws the graph on-screen and halts program execution until that window is closed. If instead you wish to save the figure to a file, use savefig('filename'), where the filename should end with the type of file you wish to save. The most common filetypes are .ps, .png, and .pdf. If you wish to save the figure *and* show the results, call savefig() before show() since show() halts program execution.

Where to learn more

Pylab is much more powerful than one would guess from the simplified description in this chapter. Pylab is also rapidly changing. Although the core functionality described here is unlikely to change significantly, more features are constantly being added. Google is your friend: searching for "matplotlib polar", for example, will tell you everything you need to know about plotting on non-Cartesian axes.

If you prefer to browse through books rather than websites, Shai Vaingast's book [17] is particularly useful.

3.3 Problems

3-0 Solve this system of equations for a through f.

$$\begin{aligned} a/5 + b + c + d - e - f &= 24.1312 \\ a + c - d + 5e + f &= 46.2798 \\ -a - 3b + 2c - d - e - f &= -61.8372 \\ 5b - c + d + 2e &= 31.1466 \\ a - 2b + 3c + d/2 &= 51.2106 \\ a/2 - 2b - 2c - d + e - 6f &= -5.7008 \end{aligned}$$

3-1 A ball is thrown upwards with initial velocity $v_o = 5$m/s and an initial height $y_o = 3$ m. Write a Python program to plot y(t) from $t = 0$ until the ball hits the ground.

3-2 A mass m is suspended from a spring of spring constant k. The mass is displaced from equilibrium by an initial distance y_o, then released. Write a Python program to plot $y(t)$ for some reasonable set of parameters.

3-3 For the previous problem, plot a *phase-space* plot. The horizontal axis should be position, and the vertical axis velocity.

3-4 Repeat the previous two problems, but add damping to the spring-mass system. In other words, the equation of motion for the system is

$$\ddot{y} = -\frac{k}{m}y - \beta\dot{y} \; .$$

Assume that $\beta < 2\sqrt{k/m}$.

Chapter 4

Ordinary Differential Equations

If you give a large wooden rabbit some initial vertical velocity v over a castle wall, you will note that the vertical component of velocity gradually decreases. Eventually the vertical velocity component becomes negative, and the wooden rabbit comes back to the ground with a speed roughly equal to the speed it had when it left.

This phenomenon of "constant-acceleration motion" was probably discussed in your introductory physics course, although less Python-oriented courses tend to use more standard projectiles. Ignoring for the moment the fact that the exact solution to constant-acceleration motion is well-known,[1] let's use numeric techniques to approximate the solution.

To find the velocity v of the projectile, we use the definition of acceleration:

$$a \equiv \frac{dv}{dt}$$

Rearranging this a bit gives us

$$dv = a\,dt$$

or since we're going to need to work with finite-sized steps,

$$\Delta v = a\Delta t$$

Knowing the initial velocity v_i and the change Δv in the velocity we can estimate the new velocity:

$$v = v_i + a\Delta t \Longrightarrow v_{i+1} = v_i + \frac{d}{dt}[v]\,\Delta t \tag{4.1}$$

[1] $v(t) = v_i + at$, $x(t) = x_i + v_i t + \frac{1}{2}at^2$

We can use an identical process to estimate the position of the projectile from its current state $\{x_i, v_i\}$ and the definition $v = \frac{dx}{dt}$:

$$x_{i+1} = x_i + \frac{d}{dt}[x]\Delta t \tag{4.2}$$

This method does not give an exact result, but for small Δt it's reasonably close. It may seem ridiculous to use it, given that we know the answer in exact form already, but for many problems we do *not* know the exact form of the answer. If we consider air resistance, for example, then the acceleration of a projectile depends on the temperature, altitude, and projectile velocity. In such a case, we may have no option other than this sort of approximation.

4.0 Euler's Method

The method we've described so far is called *Euler's method*, and it may be helpful to consider a picture of how it works. (See figure 4.0.)

Starting from point (t_1, x_1), take the slope of $x(t)$ at that point and follow the slope through the time-step Δt to find the approximate value of x_2. Repeat this process from point x_2 to find x_3, and so on.

The error shown in figure 4.0 is pretty large, but you can immediately see that the results would be better with smaller Δt. To determine the sensitivity of Euler's method to the size of Δt, start with a Taylor expansion of $x(t)$,

$$x(t + \Delta t) = x(t) + \dot{x}\Delta t + \ddot{x}\frac{\Delta t^2}{2} + \cdots \tag{4.3}$$

where $\dot{x}$ indicates a first derivative with respect to time, $\ddot{x}$ is the second derivative, and so on. The first two terms on the right side of equation 4.3 are Euler's method. The error in each step for Euler's method is on the order of Δt^2, since that's the first term omitted in the Taylor expansion. However, the number of steps is $N = \tau/\Delta t$ so the total error by the end of the process is on the order of Δt. Decreasing the size of Δt improves your result linearly.

Notice that Euler's method only works on first-order differential equations. This is not a limitation, though, because higher-order differential equations can be expanded into systems of first-order differential equations.

Example 4.0.1
The equation for damped harmonic motion is given by

$$\ddot{x} = -\omega^2 x - \beta\dot{x} \tag{4.4}$$

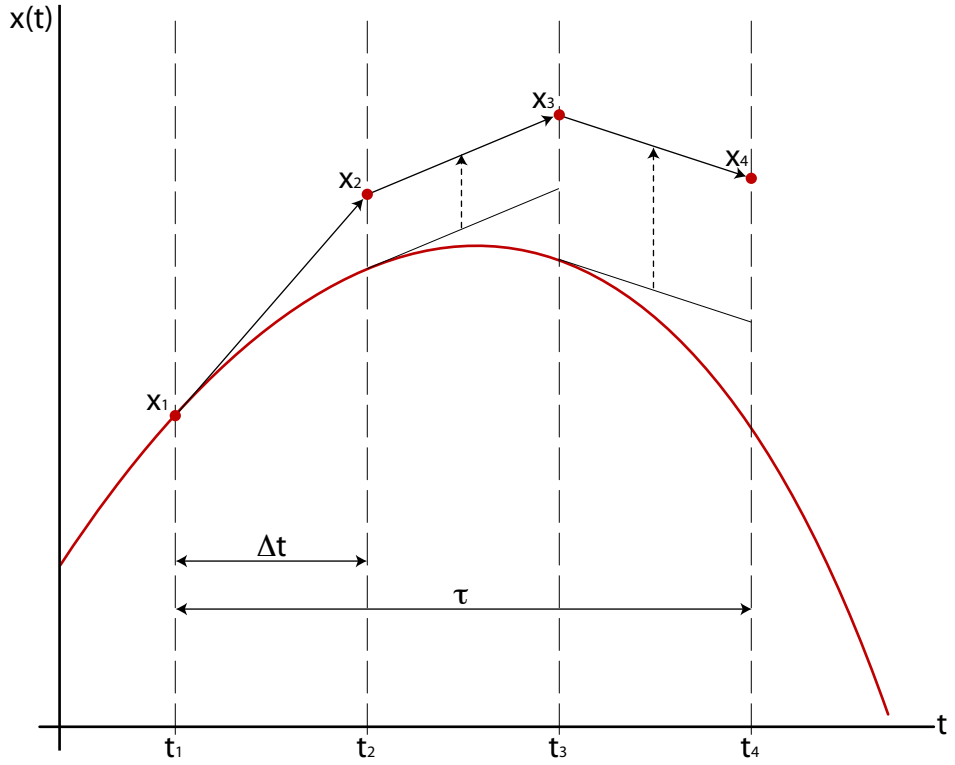


Figure 4.0: Euler's method: The slope at each point x_i is used to find the new point x_{i+1}. The smooth curve is the actual function, and the small circles are the approximation to the function determined by Euler's method.

We can express this second-order differential equation as two first-order differential equations by introducing a new variable v:

$$v \equiv \dot{x} \tag{4.5}$$

With this definition, equation 4.4 becomes

$$\dot{v} = -\omega^2 x - \beta v \tag{4.6}$$

which is a first-order differential equation.

First-order equations 4.5 and 4.6 together are equivalent to second-order equation 4.4.

4.1 Standard Method for Solving ODE's

We're going to introduce a number of methods of solving ODE's in this chapter, all of them better than Euler's method in one way or another. Before we do, though, let's take some time to develop a "standard model" for solving ODE's. That way, we can set any problem up once and then use the method of our choice to solve it, with a minimum amount of reprogramming on our part.

To start, consider the differential equation for a large wooden rabbit in free-fall:

$$\ddot{x} = -g \tag{4.7}$$

Equation 4.7 can be broken into two first-order equations:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -g \end{aligned} \tag{4.8}$$

The individual Euler-method solutions to those first-order equations are

$$\begin{aligned} x_{i+1} &= x_i + \dot{x}\Delta t \\ v_{i+1} &= v_i + \dot{v}\Delta t \end{aligned} \tag{4.9}$$

There is a symmetry in equations 4.9 that just makes you want to write them as a single vector equation:

$$y_{i+1} = y_i + \dot{y}\Delta t \tag{4.10}$$

where

$$y = \begin{bmatrix} x \\ v \end{bmatrix} \tag{4.11}$$

Equations 4.8 could be written with the same vector notation:

$$\dot{y} = \begin{bmatrix} v \\ -g \end{bmatrix} \tag{4.12}$$

Now here's the critical thing: *Equations 4.11 and 4.12 define the problem.* Equation 4.11 defines how things are arranged in the vector y, which holds everything we know about the system at some instant. Equation 4.12 defines the differential equation we're solving.

Now if only Python could do vector math correctly... Lists and tuples just don't work at all here, since multiplying a list by a number n just gives

us a list containing n copies of the original list. But the numpy package provides just this vector-math functionality. (See Chapter 3.)

This vector notation allows us to break the process into two parts: defining the problem and solving the problem. We define the problem with a function that returns the derivatives of each element in y, as in equation 4.12.

Example 4.1.1
Define a function that will return the derivatives necessary to calculate the motion of a large wooden rabbit in free-fall.

```
def FreeFall(state, time):
    """
    This function defines the ODE d^2x/dt^2 = -g.
    It takes the vector y and returns another vector
    containing the derivatives of each element in
    the current state of the system.

    The first element in state is position x, and the
    derivative of x is velocity, v. So the first
    element in the return vector is v, which is the
    second element in state.

    The second element in state is v, and the
    derivative of v is -g for free-fall.

    The result is returned as a numpy array so the
    rest of the program can do vector math.
    """
    g0 = state[1]
    g1 = -9.8
    return numpy.array([g0, g1])
```

Notice that we've passed "time" to the function in example 4.1.1, but we haven't used it. Time does not enter into the problem, since gravitational force is a constant. We've included time since we're developing a *general* method for ODE's and our method should include capability of dealing with time-dependent equations.

Now that the problem is defined, we need a general method of calculating the next "state" of the system.

Example 4.1.2
Write a general Euler's-method routine that will calculate the next state of the system from the current state, the derivatives, and the desired time step.

```
def euler(y, t, dt, derivs):
    """
    A routine that impliments Euler's method of finding
    the new 'state' of y, given the current state, time,
    and desired time step. 'derivs' must be a function
    that returns the derivatives of y and thus defines
    the differential equation.
    """
    y_next = y + derivs(y,t) * dt
    return y_next
```

Again, this routine thinks it needs to know the time, just to ensure that the general method works for time-dependent ODE's. If time does not matter, just give it *something* to keep it happy. The Euler routine will pass the time along to the derivs function, which will ignore it if it's not needed.

Note that the code in example 4.1.1 is specific to the problem at hand, but the code in example 4.1.2 is generally applicable any time we want to use Euler's method. It would be a good idea to save the definition in 4.1.2 somewhere handy so that you can just import it when needed.[2]

In the next example, let's take everything we've done so far and put it together.

Example 4.1.3
Write a program that plots the motion of a mass oscillating at the end of a spring. The force on the mass should be given by $F = -mg + kx$.

```
#!/usr/bin/env python
"""
Program to plot the motion of a mass hanging on the
end of a spring. The force on the mass is given by
F = -mg - kx.
"""
```

[2] I personally keep a file called "tools.py" that contains all of the generally-handy functions developed in this course, such as the secant method for rootfinding and Simpson's method of integrating.

```
from pylab import *
from tools import euler # Euler's method for ODE's, written earlier

N = 1000                # number of steps to take
xo = 0.0                # initial position, spring
                        # unstretched.
vo = 0.0                # initial velocity

tau = 3.0               # total time for the
                        # simulation, in seconds.
dt = tau/float(N-1)     # time step

k = 3.5                 # spring constant, in N/m
m = 0.2                 # mass, in kg
gravity = 9.8           # g, in m/s^2

# Since we're plotting vs t, we need time for that plot.
time = linspace(0, tau, N)

"""
Create a Nx2 array for storing the results of our
calculations. Each 2-element row will be used for
the state of the system at one instant, and each
instant is separated by time dt. The first element
in each row will be position, the second velocity.
"""
y = zeros([N,2])

y[0,0] = xo             # set initial state
y[0,1] = vo

def SHO(state, time):
    """
    This defines the differential equation we're
    solving: dx^2/dt = -k/m x - g.

    We break this second-order DE into two first-
    order DE's by introducing v:
    dx/dt = v
    dv/dt = k/m x - g
    """
    g0 = state[1]
    g1 = -k/m * state[0] - gravity
    return array([g0, g1])
```

```
# Now we do the calculations.
# Loop only to N-1 so that we don't run into a
# problem addressing y[N+1] on the last point.
for j in range(N-1):
    # We give the euler routine the current state
    # y[j], the time (which we don't care about
    # at all in this ODE) the time step dt, and
    # the derivatives function SHO().
    y[j+1] = euler(y[j], time[j], dt, SHO)

# That's it for calculations! Now graph the results.
# start by pulling out what we need from y.
xdata = [y[j,0] for j in range(N)]
vdata = [y[j,1] for j in range(N)]

plot(time, xdata)
plot(time, vdata)
xlabel("time")
ylabel("position, velocity")
show()
```

The output of the program in example 4.1.3 is shown in figure 4.1. Note that the position is always negative, while the velocity is either positive or negative.

4.2 Problems with Euler's Method

Euler's method is easy to understand, but it has one very large problem. Since the method approximates the solution as a linear equation, the Euler solution always underestimates the curvature of the solution. The result is that for any sort of oscillatory motion, the energy of Euler's solution increases with time.

If you look closely at figure 4.1, you can see that the maximum velocity is increasing even after just two cycles. If we change the total simulation time tau to 20 seconds in the example 4.1.3, the error is more visible: it grows exponentially as shown in figure 4.2. Increasing the number of steps improves things, (Figure 4.3) but the error is still there and still increases exponentially with each step. Even increasing the number of steps to absurd levels does not eliminate the problem: at 200,000 points there is still a slight but visible increase in peak velocity in the output of the program!

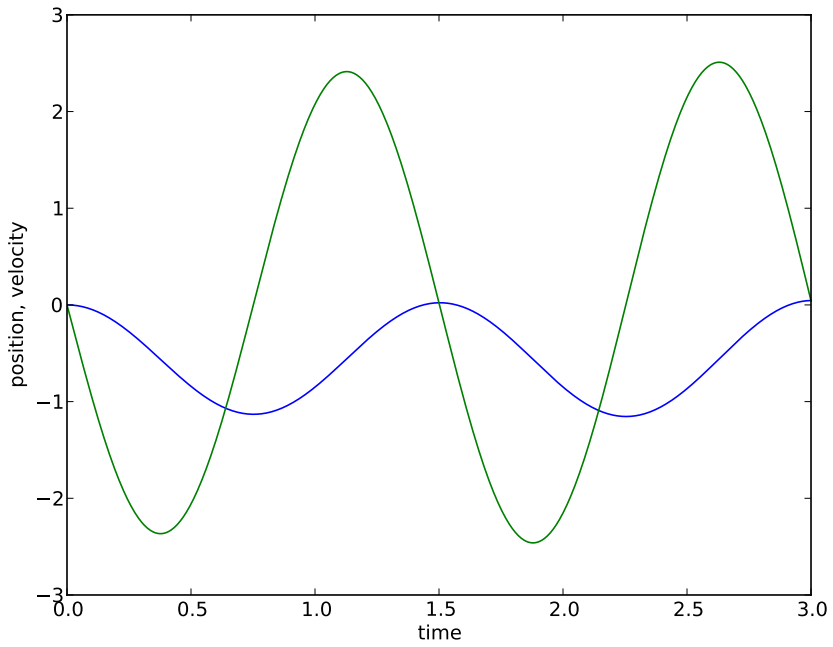


Figure 4.1: Output of program given in example 4.1.3.

4.3 Euler-Cromer Method

There is a very simple fix to Euler's method that causes it to work much better for oscillatory systems.[3] The "stock" Euler method uses the position x_i and velocity $\dot{x}_i$ at step i to find the position x_{i+1}. As it turns out, the errors exactly cancel out — for simple harmonic motion — if instead of x_i and $\dot{x}_i$ we use x_i and $\dot{x}_{i+1}$. In other words, use the current position and the *next* velocity to find the next position.

Although this is a clever fix for the current problem, we have to be a bit suspicious of it. The Euler-Cromer method gives good results for simple harmonic motion, but it's not immediately obvious that it would give the same improvment for other ODE's for which we don't already know the solution. We're better off developing methods of solving ODE's that give better results in general, rather than tweaking the broken Euler method so that it gives correct results for some particular problem.

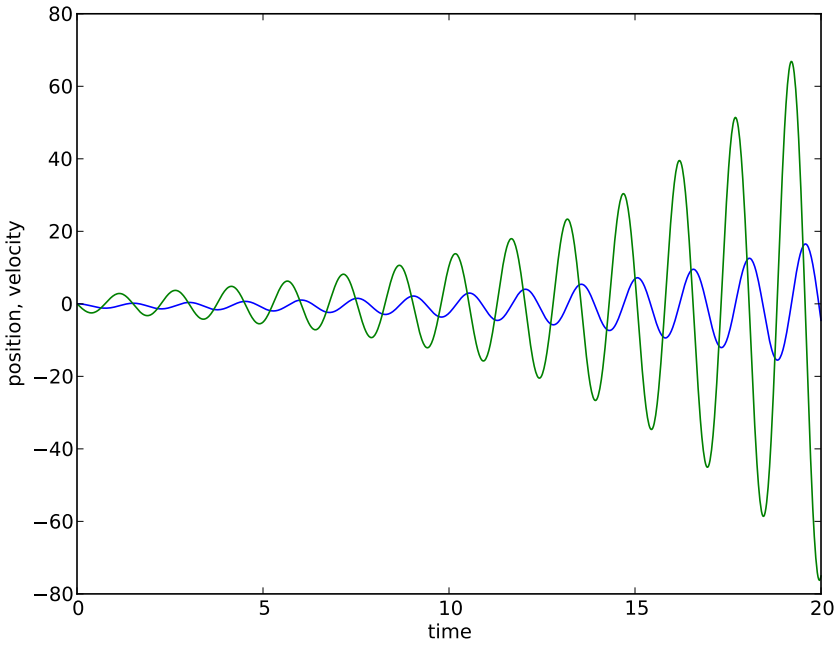


Figure 4.2: Output of program in example 4.1.3, with total time 20.0 seconds and N = 1000. Note the vertical scale!

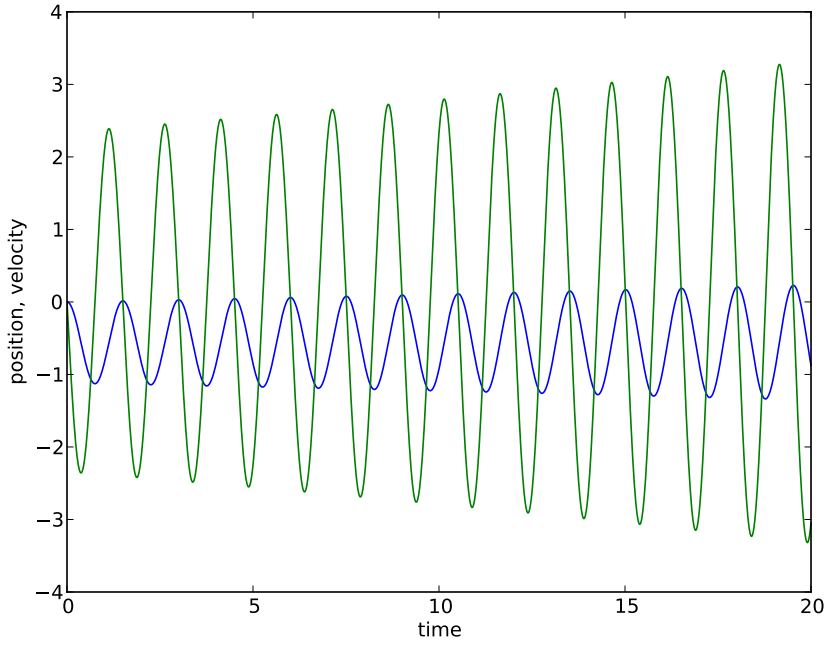


Figure 4.3: Output of program in example 4.1.3, with total time 20.0 seconds and N changed to 10,000. Despite the very large number of points in this approximation, the energy is still visibly increasing with time.

4.4 Runge-Kutta Methods

The most popular and stable general technique of solving ODE's is a set of methods known as "Runge-Kutta" methods.

I'll start the explanation of these methods with a bit of notation. We define y as the function we're looking for, and g as the derivative of y. Now g is in general a function of both y and t, so the second derivative of y is, using the chain rule,

$$\begin{aligned}\ddot{y} &= \frac{d}{dt}[\dot{y}] \\ &= \frac{d}{dt}[g(y,t)] \\ &= \frac{\partial g}{\partial t} + \frac{\partial g}{\partial y}\frac{dy}{dt} \\ &= g_t + g_y g\end{aligned}$$

where $g_\xi \equiv \frac{\partial g}{\partial \xi}$. Similarly,

$$\dddot{y} = g_{tt} + 2gg_{ty} + g^2 g_{yy} + gg_y^2 + g_t g_y$$

The Taylor expansion of $y(t+\Delta t)$ is then

$$\begin{aligned}y(t+\Delta t) = y(t) &\\ &+ g\Delta t \\ &+ \frac{\Delta t^2}{2!}(g_t + g_y g) \\ &+ \frac{\Delta t^3}{3!}(g_{tt} + 2gg_{ty} + g^2 g_{yy} + gg_y^2 + g_t g_y) \\ &+ \mathcal{O}(\Delta t^4)\end{aligned} \tag{4.13}$$

We could also expand $y(t)$ as a linear combination of some set of polynomials:

$$y(t+\Delta t) = y(t) + \alpha_1 k_1 + \alpha_2 k_2 + \cdots + \alpha_n k_n \tag{4.14}$$

where

$$\begin{aligned}
k_1 &= \Delta t g(y, t) \\
k_2 &= \Delta t g(y + \nu_{21} k_1, t + \nu_{21} \Delta t) \\
k_3 &= \Delta t g(y + \nu_{31} k_1 + \nu_{32} k_2, t + \nu_{31} \Delta t + \nu_{32} \Delta t) \\
&\vdots \\
k_n &= \Delta t g \left(y + \sum_{\ell=1}^{n-1} \nu_{n\ell} k_\ell, t + \Delta t \sum_{\ell=1}^{n-1} \nu_{n\ell} \right)
\end{aligned} \tag{4.15}$$

The α and ν are "parameters to be determined".

The next step is to expand 4.14 with a Taylor expansion, and compare the result with 4.13.

$$\begin{aligned}
y(t + \Delta t) &= y + \alpha_1 k_1 + \alpha_2 k_2 \\
&= y + \alpha_1 [\Delta t g(y, t)] + \alpha_2 [\Delta t g(y + \nu_{21} k_1), t + \nu_{21} \Delta t]
\end{aligned} \tag{4.16}$$

The first and second terms in 4.16 are exact already, so we expand only the third term:

$$\begin{aligned}
k_2 &= \Delta t [g + \nu_{21} k_1 g_y + \nu_{21} \Delta t g_t + \mathcal{O}(\Delta t^2)] \\
&= \Delta t g + \nu_{21} \Delta t^2 g g_y + \nu_{21} \Delta t^2 g_t + \mathcal{O}(\Delta t^3)
\end{aligned} \tag{4.17}$$

Substituting 4.17 into 4.16 and rearranging terms gives us

$$y(t + \Delta t) = y + [(\alpha_1 + \alpha_2) g] \Delta t + [\alpha_2 \nu_{21} (g g_y + g_t)] \Delta t^2 \tag{4.18}$$

We can compare 4.18 with 4.13 to determine our "parameters to be determined".

$$\begin{aligned}
\alpha_1 + \alpha_2 &= 1 \\
\alpha_2 \nu_{21} &= \frac{1}{2}
\end{aligned} \tag{4.19}$$

There are three unknowns in 4.19, and only two equations, so the system is underspecified. We can pick one value, then. If we choose $\nu_{21} = 1$, then α_1 and α_2 are each $\frac{1}{2}$. These values, applied to equation 4.14, give us *a* second-order Runge-Kutta method:

$$y(t + \Delta t) = y + \frac{1}{2} k_1 + \frac{1}{2} k_2 + \mathcal{O}(\Delta t^3) \tag{4.20}$$

where

$$k_1 = \Delta t g(y, t)$$
$$k_2 = \Delta t g(y + k_1, t + \Delta t)$$

Conceptually, this second-order Runge-Kutta method is equivalent to taking the *average* of the slope at t and at $t + \Delta t$ and using that average slope in Euler's method to find the new value $y(t + \Delta t)$.

The error in second-order Runge-Kutta method is on the order of Δt^3 per step, so after $N = \frac{\tau}{\Delta t}$ steps the total error is on the order of Δt^2. This total error is a significant improvement over Euler's method, even considering the roughly $2\times$ increase in computational effort. If $\frac{\Delta t}{\tau} \approx 0.01$, then the error (compared to Euler's method) goes down by about 100, for a cost increase of about 2. Alternately, one can use this method to get the *same* accuracy as with Euler's method for a fifth of the effort.

An important consideration is that the solution to 4.19 given by 4.20 is *one* of the possible second-order Runge-Kutta methods. We could just as easily set $\alpha_1 = 0$ in 4.19, in which case $\alpha_2 = 1$ and $\nu_{21} = \frac{1}{2}$. In that case, the resulting second-order method becomes

$$y(t + \Delta t) = y(t) + \Delta t g(y + \frac{1}{2}k_1, t + \frac{1}{2}\Delta t) \tag{4.21}$$

where $k_1 = \Delta t g(y, t)$. This second-order Runge-Kutta method is equivalent to taking the slope at the midpoint between t and $t + \Delta t$, and using that slope in Euler's method.

Example 4.4.1

Here is a second-order Runge-Kutta method routine:

```
def rk2(y, time, dt, derivs):
    """
    This function moves the value of 'y' forward by a single
    step of size 'dt', using a second-order Runge-Kutta
    algorithm. This particular algorithm is equivalent to
    finding the average of the slope at time t and at time
    (t+dt), and using that average slope to find the new
    value of y.
    """
    k0 = dt * derivs(y, time)
    k1 = dt * derivs(y + k0, time + dt)
    y_next = y + 0.5 * (k0 + k1)

    return y_next
```

It's a worthwhile exercise to modify the code in example 4.1.3 to use this routine instead of the Euler method and see the difference. The effort we put into splitting the definition of the differential equation from the solution method, in section 4.1, pays off now: if you add the function rk2() to your 'tools.py' file, the only code change necessary to make this switch is to import and call 'rk2' instead of 'euler'.

This second-order Runge-Kutta method comes from taking a Taylor expansion of the function we're trying to solve, and then keeping the first few terms in that expansion. As you might guess from previous sections in this book, it's possible to take more terms and get even better accuracy. Using the same procedure outlined above — but taking terms to fourth order — gives us another set of equations involving α and ν, which are underspecified just as in the case of second-order Runge-Kutta. The most commonly-used solution generates this set of equations:

$$\begin{aligned} y(t+\Delta t) &= y(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + \mathcal{O}\left(\Delta t^4\right) \\ k_1 &= g(y,t)\,\Delta t \\ k_2 &= g\left(y + \frac{1}{2}k_1, t + \frac{1}{2}\Delta t\right)\Delta t \\ k_3 &= g\left(y + \frac{1}{2}k_2, t + \frac{1}{2}\Delta t\right)\Delta t \\ k_4 &= g(y + k_3, t + \Delta t)\,\Delta t \end{aligned} \tag{4.22}$$

Converting this algorithm to a useful function is left as an exercise to the student. (It's not particularly difficult.)

There are more specialized methods of solving differential equations which can give better results for certain problems, but as a general-purpose method fourth-order Runge-Kutta offers a good combination of speed, stability, and precision. There is one significant area of improvement for this method, *adaptive stepsize*, which you'll have to learn from a different source.[13]

Here is an example of solving a differential equation in which the force changes direction periodically.

Example 4.4.2

A spring and mass system is shown in figure 4.4. The coefficient of friction μ is not negligible. Generate a position vs. time plot for the motion of the mass, given an initial displacement $x = 0.2$

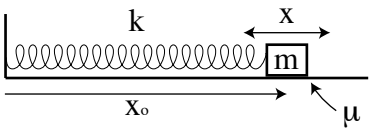


Figure 4.4: Horizontal Spring/mass system

m, spring constant $k = 42$ N/m, mass $m = 0.25$ kg, coefficient of friction $\mu = 0.15$, and initial velocity $v_i = 0$.

Most of the solution is identical to the program in example 4.1.3. There are minor changes required, such as defining μ, adjusting parameters, and changing the solving function from euler() to rk4(), but the significant change is replacing the derivative function SHO() with a new function specific to this problem.

```
#!/usr/bin/env python
"""
Program to plot the motion of a mass and spring on
a horizontal surface with friction.
F = - kx +/- mu m g
"""

from pylab import *
from tools import rk4

N = 500                     # number of steps to take
xo = 0.2                    # initial position
vo = 0.0                    # initial velocity

tau = 3.0                   # total time for the
                            # simulation, in seconds.
dt = tau/float(N-1)         # time step

k = 42.0                    # spring constant, in N/m
m = 0.25                    # mass, in kg
gravity = 9.8               # g, in m/s^2
mu = 0.15                   # friction coefficient

"""
Create a Nx2 array for storing the results of our
calculations. Each 2-element row will be used for
```

```
    the state of the system at one instant, and each
    instant is separated by time dt. The first element
    in each row will be position, the second velocity.
    """
    y = zeros([N,2])

    y[0,0] = xo                 # set initial state
    y[0,1] = vo

    def SpringMass(state, time):
        """
        This defines the differential equation we're
        solving: dx^2/dt = -k/m x +/- mu g.

        We break this second-order DE into two first-
        order DE's by introducing v:
        dx/dt = v
        dv/dt = k/m x +/- mu g

        Note that the direction of the frictional
        force changes, depending on the sign of v.
        We handle this with an if statement.
        """
        g0 = state[1]

        if g0 > 0:
            g1 = -k/m * state[0] - gravity * mu
        else:
            g1 = -k/m * state[0] + gravity * mu

        return array([g0, g1])

    # Now we do the calculations.
    # Loop only to N-1 so that we don't run into a
    # problem addressing y[N+1] on the last point.
    for j in range(N-1):
        # We give the euler routine the current state
        # y[j], the time (which we don't care about
        # at all in this ODE) the time step dt, and
        # the derivatives function SHO().
        y[j+1] = rk4(y[j], 0, dt, SpringMass)

    # That's it for calculations! Now graph the results.
    time = linspace(0, tau, N)
    plot(time, y[:,0], 'b-', label="position")
```

```
xlabel("time")
ylabel("position")
show()
```

Notice that the derivs() function can contain anything we need to make things work. In this case, the function uses an **if** statement to change the direction of the frictional force as needed. The graphical output of the complete program is shown in figure 4.5.

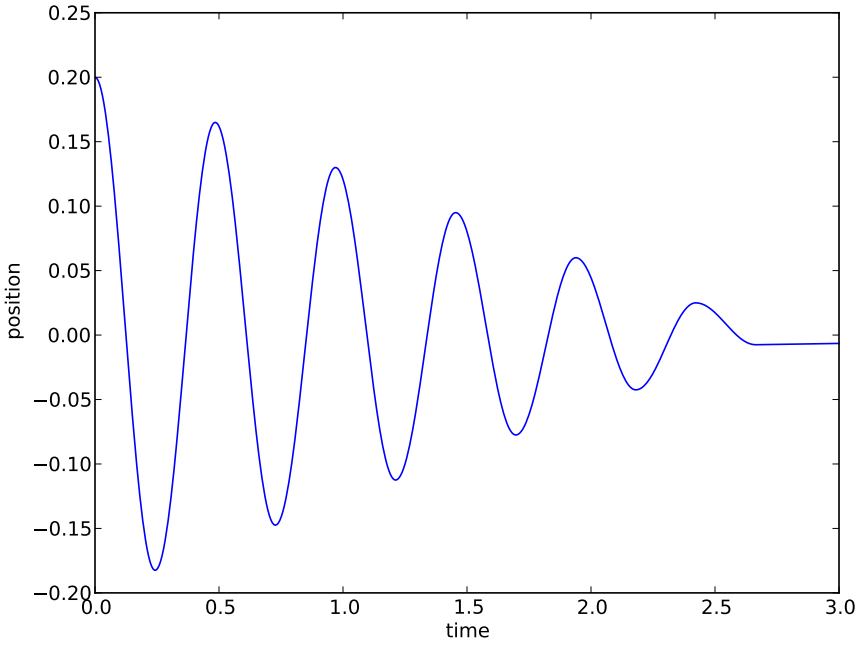


Figure 4.5: Motion of mass shown in figure 4.4

There is one more level of improvement we can make to our ODE-solving method. So far we've dealt with individual steps, but in most computational work it's more common to find solutions for the differential equation at an array of times. For example, in the previous problem we could have defined an array of times for plotting (using linspace()) and then called a function that used our ODE solver method of choice to find solutions at those times. Development of such a function is left as an exercise.

4.5 Scipy

As you might expect, Scipy has a routine that solves differential equations. This function is avalable as scipy.integrate.odeint(), and it uses variable step-sizes and error-checking methods to return very precise results, efficiently. Call this routine with a derivs function, an initial state value (which may be an array, as usual) and an array of times (rather than a time step.) The odeint() function will return an array of state values corresponding to those times.

Let's use odeint() to look at a more complex example:

Example 4.5.1
A mass m is attached to a spring with spring constant k, which is attached to a support point as shown in figure 4.6. The length of the resulting pendulum at any given time is the spring rest-length x_o plus the stretch (or compression) x, and the angle of the pendulum with respect to the vertical is θ. This is an example

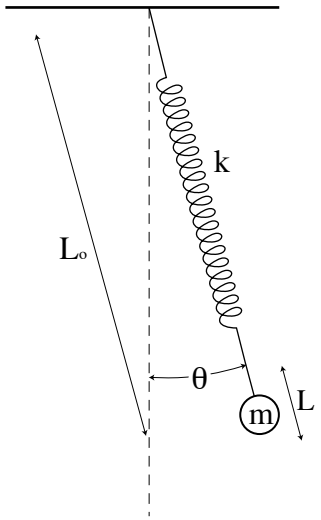


Figure 4.6: Springy Pendulum for example 4.5.1

of a coupled oscillator system: the "pendulum" oscillations in θ interact with the "spring" oscillations in x, producing a complex

mix of both. The differential equations for this system are given by[3]

$$\ddot{L} = (L_o + L)\dot{\theta}^2 - \frac{k}{m}L + g\cos\theta$$
$$\ddot{\theta} = -\frac{1}{L_o + L}\left[g\sin\theta + 2\dot{L}\dot{\theta}\right]$$

Write a program that plots the motion of the mass for some initial $\theta \neq 0$.

```
#!/usr/bin/env python
"""
Program to plot the motion of a "springy pendulum".
"""

from pylab import *
from scipy.integrate import odeint

N = 1000                    # number of steps to take

"""
We actually have FOUR parameters to track, here:
    L, L_dot, theta, and theta_dot.
So instead of the usual Nx2 array, make it Nx4.
Each 4-element row will be used for the state of
the system at one instant, and each instant is
separated by time dt. I'll use the order given above.
"""
y = zeros([4])

L_o = 1.0                   # unstretched spring length
L = 1.0                     # Initial stretch of spring
v_o = 0.0                   # initial velocity
theta_o = 0.3               # radians
omega_o = 0.0               # initial angular velocity

y[0] = L                    # set initial state
y[1] = v_o
y[2] = theta_o
y[3] = omega_o

time = linspace(0, 25, N)
```

[3]The easiest way of deriving these is via Lagrangian Dynamics, see [7].

```
k = 3.5                    # spring constant, in N/m
m = 0.2                    # mass, in kg
gravity = 9.8              # g, in m/s^2

def spring_pendulum(y, time):
    """
    This defines the set of differential equations
    we are solving. Note that there are more than
    just the usual two derivatives!
    """
    g0 = y[1]
    g1 = (L_o+y[0])*y[3]*y[3] - k/m*y[0] + gravity*cos(y[2])
    g2 = y[3]
    g3 = -(gravity*sin(y[2]) + 2.0*y[1]*y[3])/(L_o + y[0])

    return array([g0, g1, g2, g3])

# Now we do the calculations.
answer = odeint(spring_pendulum, y, time)

# Now graph the results.
# rather than graph in terms of t, I'm going
# to graph the track the mass takes in 2D.
# This will require that I change L,theta data
# to x,y data.
xdata = ( L_o + answer[:,0])*sin(answer[:,2])
ydata = -(L_o + answer[:,0])*cos(answer[:,2])

plot(xdata,ydata, 'r-')
xlabel("Horizontal position")
ylabel("Vertical position")
show()
```

Note that in previous examples the size of our state vector y has always been 2, but there's nothing magic about that. We can make it contain as many (or as few) parameters as necessary to solve the problem.

The path of the pendulum for one set of initial conditions is shown in figure 4.7. Changing the parameters results in very interesting paths — feel free to explore the results with different m, k, and initial conditions!

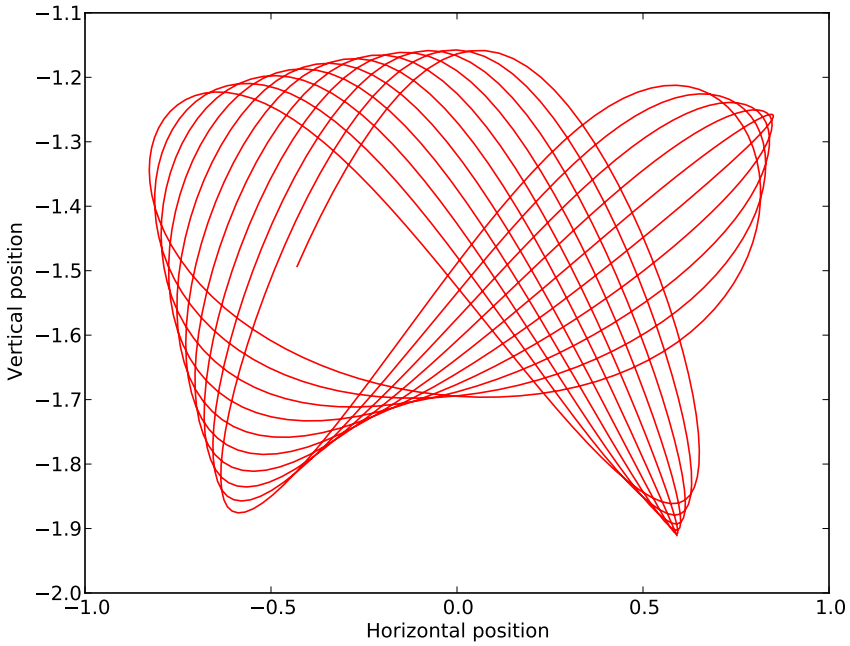


Figure 4.7: Path of the spring-pendulum in example 4.5.1, with initial conditions given in the program listing.

Let's look at one more example, this one with a time-dependent differential equation.

Example 4.5.2
Write a "derivs" function for a damped driven pendulum:

$$\ddot{\theta} = -\frac{g}{L}\sin\theta - b\dot{\theta} + \beta\cos\omega t$$

```
def pendulum_def(state, time):
    """
    The state variable should be defined in our
    usual way: y[0] = theta, y[1] = d(theta)/dt.
    The constants "gravity", "length", "beta",
    "b", and "omega" are assumed to be defined
```

```
    elsewhere.
    """
    g0 = y[1]
    g1 = -gravity/length * sin(y[0]) - b*y[1] +
         beta*cos(omega * time)
```

Incorporating time into our solving mechanism is easy, since we've included the *capacity* to use time since we started this "standard method".

4.6 Problems

4-0 Express each of these differential equations as a set of first-order differential equations

(a)
$$m\ddot{x} = f(x,t)$$

(b)
$$A\ddot{x} + B\dot{x} + Cx = D$$

(c)
$$m\ddot{\theta} = -\sqrt{\frac{g}{L}}\sin\theta - \beta\dot{\theta} + \gamma\sin\omega t$$

(d)
$$\dddot{y} + \ddot{y} + \dot{y} + y = 0$$

4-1 The number of radioactive atoms that decay in a given time period is proportional to the number of atoms in the sample:

$$\frac{dN}{dt} = -\lambda N$$

Write a program that uses Euler's method to plot $N(t)$. Have your program also plot the exact solution, $N(t) = N_o e^{-\lambda t}$, for comparison.

4-2 Write an appropriate "derivs" function for each of the differential equations in problem 0.

4-3 Write a function that impliments the 4^{th}-order Runge-Kutta method described in equation 4.22. Make sure it works by testing it for some previous problem or example that used Euler's method or second-order Runge-Kutta.

4-4 Write a function that solves ODEs for each point on an array of time values, such as is described on page 100. You may use whatever individual-step routine you wish (Euler, 4^{th}-order Runge-Kutta, etc.) Test your routine on a differential equation with a known solution, and show that it works.

4-5 In a radioactive decay chain, element A decays to element B which decays to C, and so on until the decay chain reaches a stable element. One example of such a chain is ^{90}Sr, which decays to ^{90}Y, which decays

to ^{90}Zr. The half-life of ^{90}Sr is 28.78 years, and the half-life of ^{90}Y is 2.67 days. ^{90}Zr is stable. This decay chain can be described by the following differential equations:

$$\frac{dN_{Sr}}{dt} = -\lambda_{Sr} N_{Sr}$$

$$\frac{dN_Y}{dt} = -\lambda_Y N_Y - \frac{dN_{Sr}}{dt}$$

Plot the relative activity of a sample of ^{90}Sr as a function of time. (A logrithmic time scale will be helpful.)

4-6 A set of magnetically-coupled rotors (see [10]) with velocity-dependent damping move according to the equations

$$\ddot{\theta}_1 = \beta \sin(\theta_2 - \theta_1) - b\dot{\theta}_1$$
$$\ddot{\theta}_2 = \beta \sin(\theta_1 - \theta_2) - b\dot{\theta}_2$$

Write a program that plots θ_1 and θ_2 vs. time for a total time $\tau = 20$ seconds, with $b = 0.1$ and $\beta = 5.0$.

4-7 Redo problem 6 with damping that depends on $\dot{\theta}^2$ instead of on $\dot{\theta}$. Be careful about the sign of the damping term! You may also have to change the size of b to obtain interesting results.

4-8 A spring-mass system with friction, such as that in figure 4.4, is driven by an external force $F(t) = A\cos\omega t$.

a) Plot the frequency response of this system. In other words, for some constant value of A, what is the resulting amplitude of oscillation of the mass as a function of the driving frequency ω? Choose your parameters so as to span the "interesting" frequency region, and note that a logrithmic graph axis might be helpful on one or both axes.

b) How does the frequency response change with driving amplitude A? Write a program to make a graph that answers the question.

It is possible to write a single program that answers both parts of this question with one graph, although you may find that 2-dimensional graphs are somewhat limiting for this purpose.

4-9 The rate at which a finite resource (such as oil) is gathered depends on the difficulty and on the demand. Assume that the demand D increases with the amount extracted E:

$$\frac{d}{dt}[D] = \alpha E$$

Assume that the difficulty of extraction W is inversely dependent on the fraction remaining:

$$W = \frac{1}{1 - E}$$

Finally, assume that the rate of extraction R is the ratio of demand to difficulty:

$$R = \frac{d}{dt}[E] = \frac{D}{W}$$

Assume also that the cost C of the resource depends on the extraction rate and on the demand:

$$C = \frac{D}{R}\xi$$

where ξ is some scale factor.

Write a program that plots the extraction rate and the cost as a function of time, for this simple model. Good starting values are $D_o = 0.1$, $\alpha = 2.0$, and $\xi = 0.01$, plotted over a total time range of 0 to 5.

Chapter 5

Chaos

Let's take a break from learning new computer techniques and algorithms for a bit, and spend some time *using* what we've learned so far to investigate something interesting. We'll start with something familiar: the simple pendulum.

A simple pendulum, as you probably remember from introductory physics, consists of a point mass on a massless string as shown in figure 5.0. If the

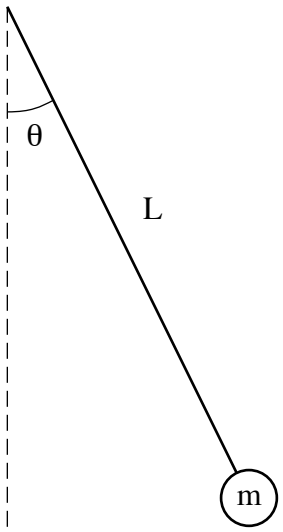


Figure 5.0: A simple pendulum

mass is released from some initial angle θ_o, the torque restoring the pendulum to the center is

$$\tau = I\alpha = -mgL\sin\theta$$

The rotational inertia is $I = mL^2$, so the equation of motion for this pendulum is

$$\ddot{\theta} = -\frac{g}{L}\sin\theta \tag{5.1}$$

At this point in your introductory physics class, you almost certainly were told to keep the angle θ small, in which case $\sin\theta \approx \theta$. For this approximation,

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\theta$$

which is simple harmonic motion, with a known solution $\theta(t) = \theta_o \cos\left(\sqrt{\frac{g}{L}}t\right)$.

5.0 The Real Pendulum

The *reason* that the small-angle equation is used in equation 5.1 is that otherwise we can't solve the equation analytically. But for this course we don't care about solving the equation analytically: numeric solutions are just fine. A "derivs" functions for equation 5.1 could look something like this:

```
def pendulum(state, time):
    """
    Differential equation for an undamped pendulum.
    state[0] should be angular position, state[1]
    should be angular velocity.
    """
    g0 = state[1]
    g1 = -gravity/length * sin(state[0])
    return numpy.array([g0,g1])
```

If we plotted θ vs time, we'd get a graph something like figure 5.1. You can see in figure 5.1 that the small-angle approximation gives a noticeably different result from the numeric solution to equation 5.1, even at this relatively small initial angle of 30°. At larger angles, the small-angle approximation becomes progressively worse.

To make our pendulum more realistic, let's add a damping term to equation 5.1.

$$\ddot{\theta} = -\frac{g}{L}\sin\theta - \beta\dot{\theta} \tag{5.2}$$

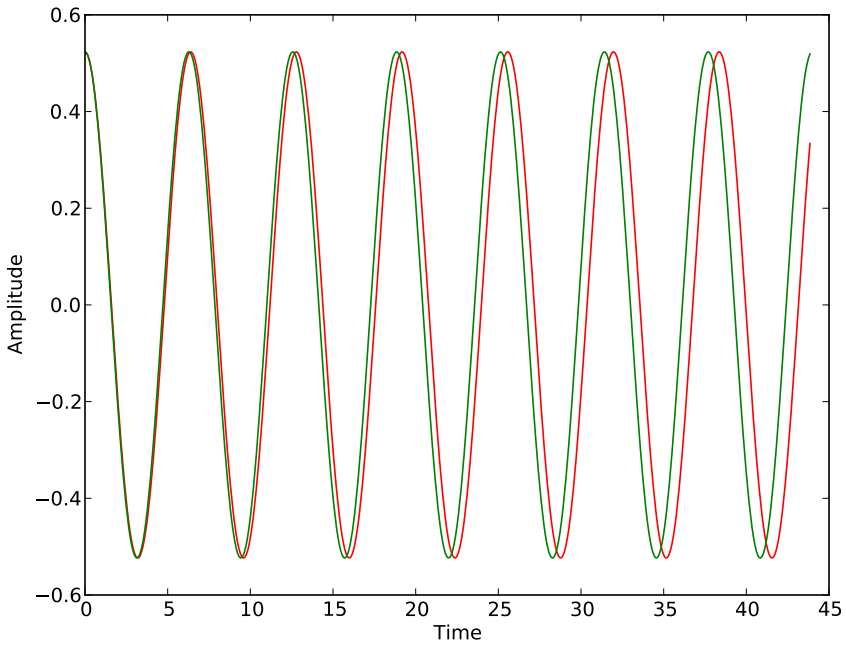


Figure 5.1: Angular position vs time for a simple pendulum. The red curve is obtained using scipy.integrate.odeint() on equation 5.1, the green curve is the result of the small-angle approximation. $\frac{g}{L} = 1$.

This damping depends on the angular velocity $\dot{\theta}$, which would be consistent with viscous damping or magnetic braking. The modified "derivs" function would look like this:

```
def pendulum(state, time):
    """
    Differential equation for a damped pendulum.
    state[0] should be angular position, state[1]
    should be angular velocity.
    gravity, length, and damping parameter beta
    are global variables.
    """
    g0 = state[1]
    g1 = -gravity/length * sin(state[0]) - beta*state[1]
    return numpy.array([g0,g1])
```

And the resulting graph of position vs time would be as shown in figure 5.2.

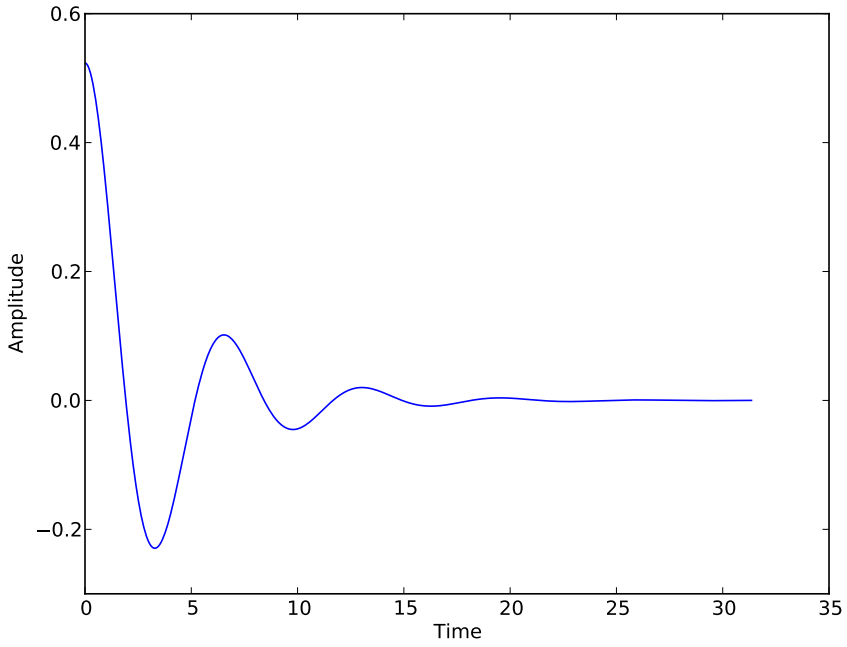


Figure 5.2: Damped simple pendulum, with $\frac{g}{L} = 1$ and damping parameter $\beta = 0.5$.

This motion is more realistic, but eventually everything just stops, which is boring. Let's add in a driving term as well, just to keep things going.

$$\ddot{\theta} = -\frac{g}{L}\sin\theta - \beta\dot{\theta} + A\cos(\omega t) \tag{5.3}$$

This equation now represents a pendulum with a time-varying torque with strength A and driving frequency ω. It's trivial to add this term to our previous "dervis" definition for the pendulum, and the resulting graph looks like figure 5.3. As you can see, the pendulum quickly reaches a steady-state motion. On the way *to* that steady-state motion, however, it actually loops completely over the top so its new equilibrium position is around $\theta = 2\pi$ rather than around $\theta = 0$. Zero and 2π are exactly the same place, so it's

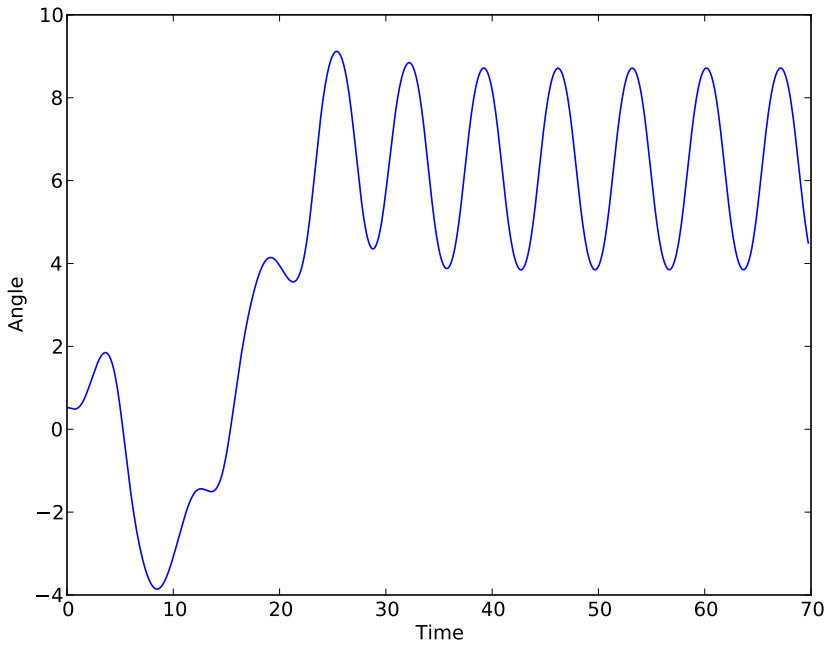


Figure 5.3: A damped, driven pendulum. Here $A = 1.5$, $\omega = 0.9$, and $\beta = 0.5$.

helpful to "wrap" the data: any time the data is more than π (or less than $-\pi$) we subtract (add) 2π so that the data range is limited to the range $\pm\pi$. It's also easiest at this point to switch the plot from lines to dots so that you don't get lines across the graph any time the pendulum loops around: see figure 5.4.

5.1 Phase Space

Phase plots are a tool often used to gain more insight into the behavior of oscillatory systems such as the driven pendulum. Instead of plotting one characteristic of the system vs. time, a phase plot is a plot of one characteristic of the system vs. another characteristic. The most common type of phase plot is velocity vs. position. Figure 5.5 shows the same motion as in figure 5.4, but as a phase plot instead of as a time plot.

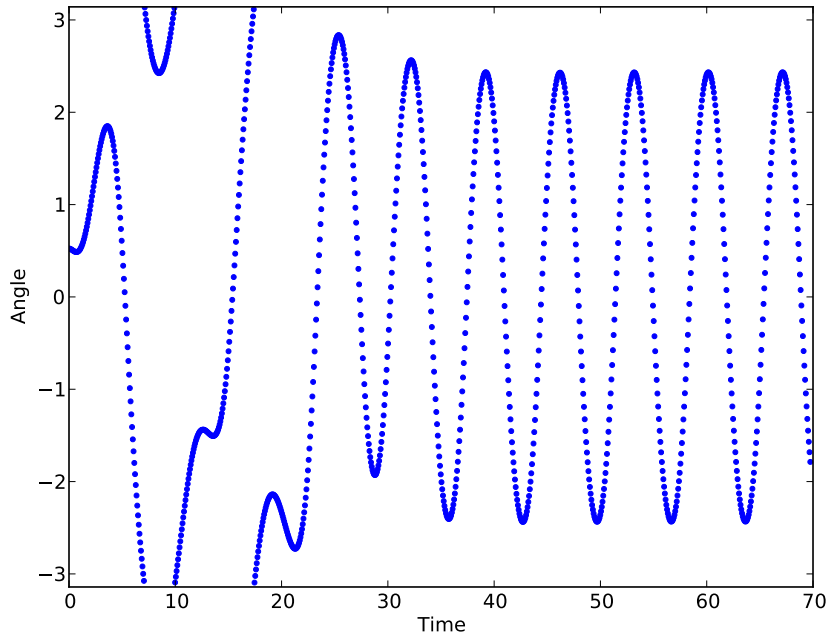


Figure 5.4: Same calculation as for figure 5.3, but with data "wrapped" to the range $\{-\pi : \pi\}$

Although a phase plot lacks the time information of the type of plots we've been using so far, it gives additional insight into the overall long-term behavior of the system. For example, if we just look at the data *after* the initial "transient" behavior dies out (after $t = 40$ in figure 5.5, for example) we see the *attractor* for this system. (figure 5.6.)

The attractor is the motion at which the system eventually arrives, independent of its starting conditions. As a trivial example, consider the undriven pendulum, figure 5.2. Regardless of the initial conditions, the pendulum eventually reaches the state (0,0) on the phase plot: hanging at $\theta = 0$, with angular velocity $\dot{\theta} = 0$. The attractor in figure 5.6 is more complicated, but not by much.

As we lower the drive frequency, we reach a frequency at which the attractor splits into a double loop instead of a single loop. We call this a "period-2" oscillation: the period of the oscillation is twice the period of the

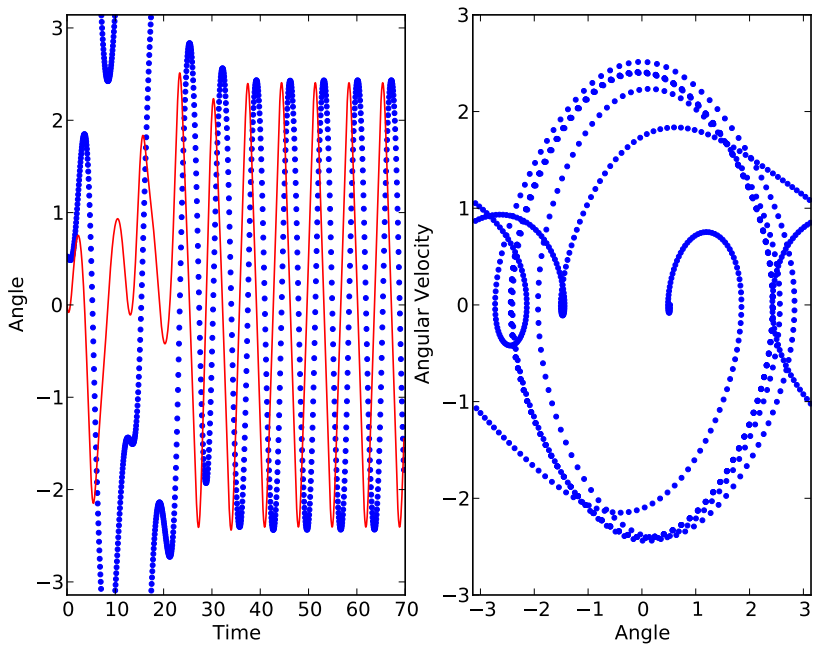


Figure 5.5: Plots of position (and velocity) vs. time, with corresponding phase plot. These two plots show the same system, undergoing the same motion.

driving force. (See figure 5.7b.) This process continues with exponentially-rapid splits into period-4 (figure 5.7c), period-8, and so on until at some frequency (see figure 5.7d) the motion is *chaotic*. Note that the initial transient behavior has been omitted from figure 5.7: what is shown is just the attractor. Figure 5.7d is a *strange attractor*.

It is very important to understand that the motion shown in figure 5.7d is not *random*. It repeats itself, but the period of that repetition may be longer than the age of the universe! The motion is still deterministic: if the pendulum is re-started in the exact same conditions, it will undergo the exact same motion. But if the pendulum is started in even slightly different conditions, the motion will be different although the attractor will be the same. This *sensitive dependence on initial conditions* is the hallmark of chaotic behavior.

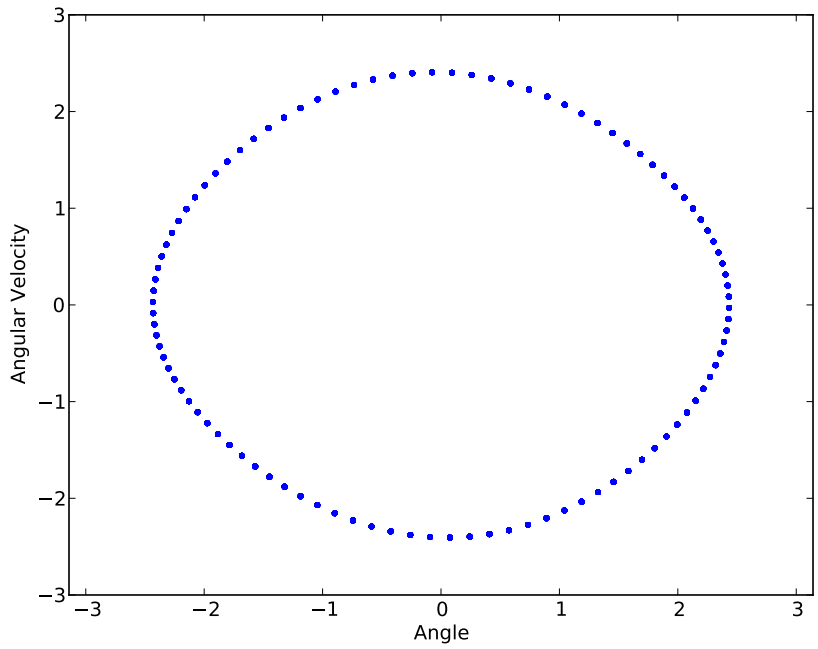


Figure 5.6: Attractor for the pendulum in figure 5.3.

5.2 Poincaré Plots

As helpful as phase space is, it still can overwhelm us with the volume of data. Another method that is helpful in analyzing the behavior of chaotic systems is to take a "snapshot" of the system once per drive cycle. Each snapshot datum is plotted as a single point in a standard phase space plot. This type of analysis is called a Poincaré plot.

For the phase space plot in figure 5.7a, the corresponding Poincaré plot would be a single point located somewhere on the stable loop of the phase space plot. The exact location of that single point would depend on *when* in the drive phase the snapshots were taken, of course: the Poincaré plot in figure 5.8a shows a Poincaré plot for the phase plot in figure 5.7a with a phase delay of half the drive period, overlaid on the corresponding phase plot. Figures 5.8b–d show corresponding Poincaré plots for figures 5.7b–5.7d. These Poincaré plots make it easy to see the period doubling that

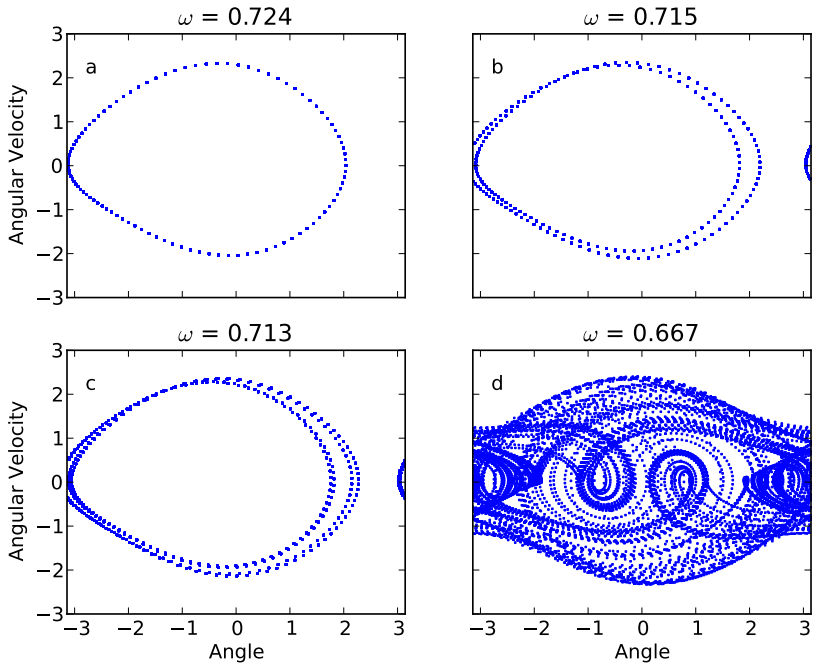


Figure 5.7: Period-doubling in a driven damped pendulum.

leads to the chaotic motion.

To help understand the meaning of the Poincaré plot, keep in mind that the system will be *somewhere* on the Poincaré plot at that moment in the drive cycle. In the case of chaotic motion, the Poincaré plot can be a quite complicated curve; but once again *chaotic* motion is not *random* motion and although we don't know exactly where on the Poincaré curve the motion will be we can say with absolute certainty that it will *not* be on a point *off* that curve.

For chaotic motion, the Poincaré curve is generally a fractal curve: the greater the magnification, the more detail becomes visible.

A program that calculates and plots the motion of this chaotic pendulum, with a Poincaré plot overlaid on the data, is given below.

```
#!/usr/bin/env python
"""
Pendulum plotting program (linear space)
```

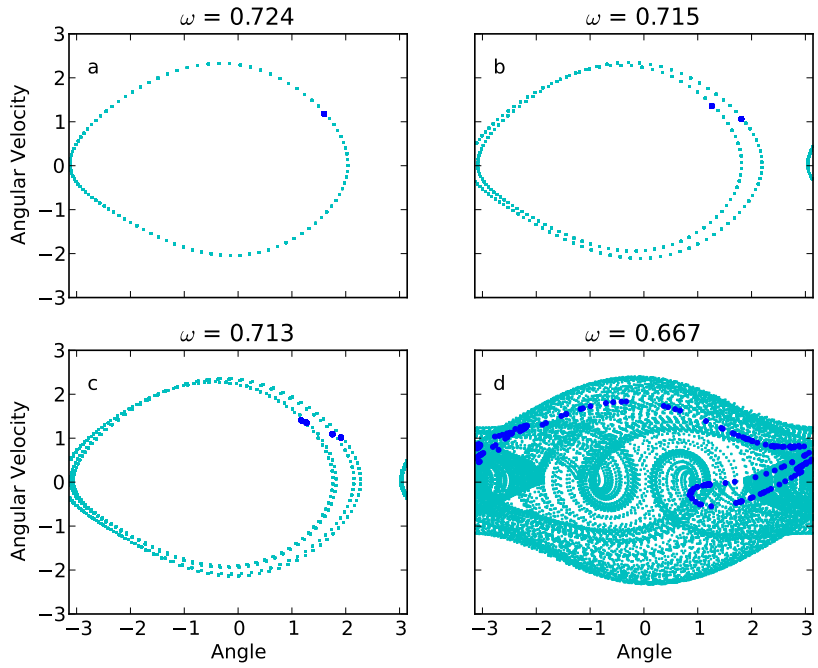


Figure 5.8: Poincaré plots corresponding to figure 5.7

```
Eric Ayars

Plots phase plot and Poincare plot for a damped driven pendulum in
chaotic motion.

The array 'y' in this program contains the position and velocity info:
    y[0] = theta
    y[1] = omega = d theta / dt
"""

from pylab import *
from scipy.integrate import odeint

steps = 100                          # steps per lap
theta_initial = pi/6.0               # initial position
v_initial = 0.0                      # initial angular velocity
#omega = 1.0                         # natural frequency
```

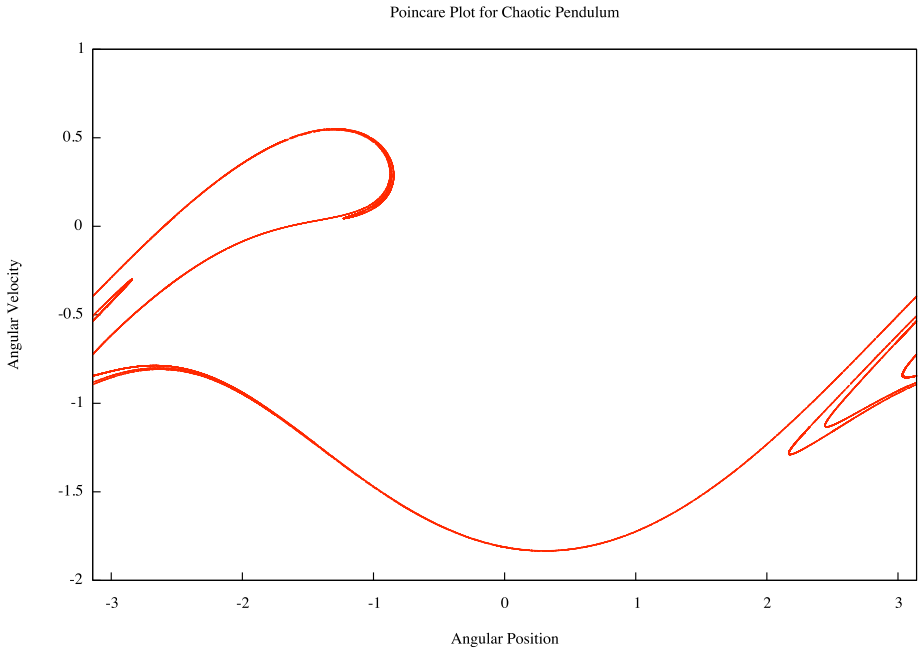


Figure 5.9: Poincaré plot for the damped driven pendulum in chaotic motion, at phase angle 0.

```
# actually, just insist that omega = 1. leave it out of calculations below.
beta = 0.5                                  # damping factor
amp = 1.2                                   # Driving torque amplitude
omega_d = 0.713                             # Driving angular frequency
time_step = 2.0*pi/(omega_d*steps)

N = 300                                     # number of drive laps to calculate
skip = 100                                  # don't plot these, let transient die.

# set up initial conditions
initial_state = array([theta_initial, v_initial])
time = arange(0.0, N*(2.0*pi)/omega_d, time_step)

def DE_pendulum(y, time):
    """
    "derivs" function for our pendulum.
    """
    g0 = y[1]
```

```
    g1 = -sin(y[0]) - beta*y[1] + amp*sin(omega_d*time)
    return array([g0, g1])

# Calculate the solutions
answer = odeint(DE_pendulum, initial_state, time)
answer = answer[skip*steps:]

# Wrap the solutions into range -pi..pi
for i, position in enumerate(answer[:,0]):
    while position > pi:
        position = position - (2.0 * pi)
        # There --- that should take care of too-large angles.
    while position < -pi:
        position = position + (2.0 * pi)
        # And that should take care of too-small angles.
    # now replace the value in answer with this wrapped version
    answer[i,0] = position

# Pull out one "snapshot" per lap for Poincare plot
offset = 50                          # phase of Poincare plot, 0-99.
max_index = (N-skip)*steps - offset # Don't try going past here.

P_thetas = []                        # build up these lists
P_omegas = []

for j in range(offset, max_index, steps):
    # This loop pulls out a snapshot of the system once per drive cycle.
    # It adds this snapshot to the lists x_thetas and x_omegas for
    # a later Poincare plot.
    P_thetas.append(answer[j,0])
    P_omegas.append(answer[j,1])

# now plot the results
plot(answer[:,0], answer[:,1], 'c,')
plot(P_thetas, P_omegas, 'b.')
xlabel('Angle')
ylabel('Angular Velocity')
xlim(-pi, pi)
show()
```

5.3 Problems

5-0 The Duffing oscillator is given by

$$\ddot{x} + \delta\dot{x} + \beta x + \alpha x^3 = \gamma \cos \omega t$$

where δ is a damping constant, $\delta \geq 0$.

Write a program to show a Poincaré section for the Duffing oscillator in a chaotic regime such as is located at $\{\alpha = 1, \beta = -1, \delta = 0.2, \gamma = 0.3, \omega = 1\}$.

Chapter 6

Monte Carlo Techniques

Monte Carlo is a small town on the French Riviera most famous for its casino. Because of this fame in the field of random events, the name "Monte Carlo techniques" is given to random (or pseudo-random) methods of solving problems on a computer.

Here's an example. Say you have an odd shape, and you want to find its area. One Monte Carlo technique for finding this area would be to surround the shape with a shape of known area (a rectangle, perhaps) and then generate a large number of random points within the known area. The area of the shape is (approximately) the fraction of the points that fall within the shape, times the known area surrounding the shape. (See figure 6.0.)

For this method to work, it's necessary that our random numbers meet three criteria:

1) They must be uniformly distributed. Obviously, if the points were distributed about some normal curve, then we'd see more points in the central area of figure 6.0 and we'd get a poor answer.

2) They must be uncorrelated. For example, if our random numbers in figure 6.0 were correlated in such a way that a low one was most often followed by a high one, and vice-versa, then the points in the figure would trend towards the upper left and lower right of the figure, and this would skew our results.

3) We need a *lot* of points. Our percent error by this method is going to scale roughly as $\frac{1}{\sqrt{N}}$, so for a mere three significant figures in the answer, we need a million data pairs.

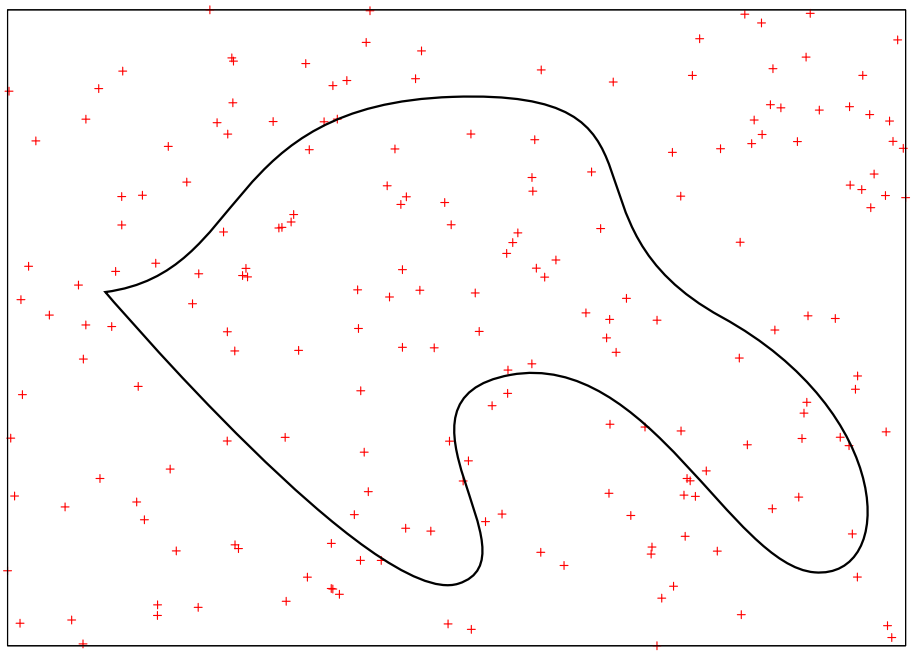


Figure 6.0: A Monte Carlo method of finding the area of an odd-shaped figure. The area of the shape is approximately 71/200 × the area of the bounding rectangle.

6.0 Random Numbers

Generating random numbers is inherently difficult. For starters, there's the question of what is random in the first place! (See figure 6.1.) Add to this the problem that people generally can't create randomness on their own. If you asked the average student to draw 200 random dots on a rectangle, they'd end up much more evenly separated than the points on figure 6.0. Adding to the difficulty with Monte Carlo techniques is the issue of trying to generate something random with computers, which are inherently non-random devices. There are physical sources of random numbers, such as the time between decay events in a radioactive sample, but it's hard to collect enough of these fast enough for what we need.

We generally give up on truly random numbers, and use "Pseudo-Random" sequences of numbers instead. If the sequence of numbers meets the criteria on page 123, then they'll do the job even if they aren't technically random.

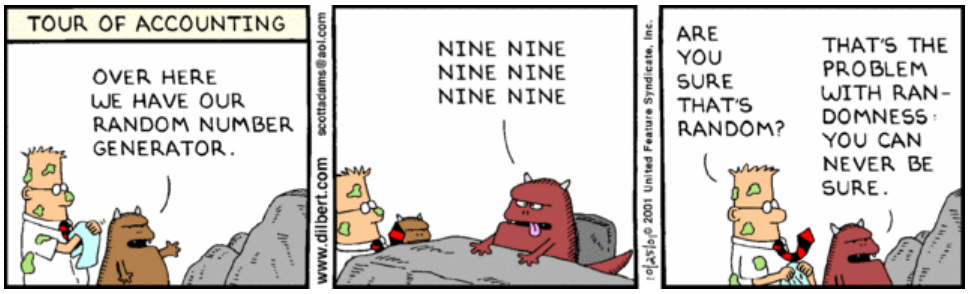


Figure 6.1: ©Scott Adams

One example of a pseudo-random number generator is the linear congruential generator:

$$x_{n+1} = (ax_n + b) \mod c \tag{6.1}$$

This pseudo-random number generator takes a "seed" x_0 and generates a series of numbers which, depending on the choices of a, b, and c, can meet the necessary criteria for Monte Carlo techniques.

Example 6.0.1
Let $a = 5$, $b = 3$, and $c = 8$. For a starting "seed" $x_0 = 1$, this generates the sequence 0, 3, 2, 5, 4, 7, 6, 1. At this point, the sequence repeats itself.

As you can see in example 6.0.1, c needs to be fairly large in order to generate a long sequence: in fact the longest possible sequence for the linear congruential generator is of length c.

One set of parameters that was used heavily in the 70's and early 80's is "Randu": $a = 65539$, $b = 0$, and $c = 2^{31}$. These parameters have the advantage of being easy for the computer to calculate, so it was possible to generate large sequences quickly. There's a problem, though, which was not discovered until much later: the Randu sequence is highly correlated! If you use Randu to plot random points in three dimensions, every point falls on one of 15 parallel planes. As you may well imagine, this belated discovery has cast doubt on many previously-calculated Monte Carlo results, and people are still trying to evaluate the damage.

Python comes with a pseudo-random number generator called the "Merseinne Twister". This is a very fast generator, and it has been extensively

tested by the mathematical community and given a clean bill of health.[1] Rather than use our own pseudo-random number generator and risk pitfalls such as Randu, we'll use this one.

To access these random-number routines, import the "random" package:

```
import random

random.random()          # generates a random float between 0 and 1
random.uniform(a,b)      # generates a random float on the range [a,b).
random.choice(list)      # returns a random element from the list
random.gauss(mu,sigma)   # returns points with gaussian distribution
                         # centered on mu, with width sigma
random.randint(a,b)      # random integer on range [a,b]
```

These are the most useful functions within random, at least for the purposes of this course. There are other functions as well: see the Python documentation.

6.1 Integration

Let's go back now and take a closer look at the first integration example in this chapter. The first question you should be asking is, "Why would anyone do this?" After all, in order for the program to determine whether a random point is inside or outside of the shape, the program will have to have an equation for the shape. If the program has an equation for the shape, shouldn't it be possible to find the area by numerically integrating, using the techniques from section 2.1? And shouldn't direct numerical integration be faster, by several orders of magnitude, than randomly selecting several million points?

These objections are entirely valid, for problems with low dimensionality. At higher dimensions, though, direct numerical integration becomes more difficult. The number of function calculations for numeric integration grows exponentially with dimension. Monte Carlo integrations also increase in difficulty with dimension, but the increase is linear. This means that an integral that takes 10 function calls to calculate in one dimension would take on the order of 10^{12} function calls in 12 dimensions. The same integrals might take a million function calls in one dimension for Monte Carlo integration, and a mere 12 million in 12 dimensions. Monte Carlo wins by a large margin.

[1] The only caveat is that the Merseinne Twister is not appropriate for cryptographic use, as it is possible to calculate the exact state of the system given a large enough sample of the output. This makes no difference to this course!

"But where," you may well ask, "does one get a 12-dimensional integral?" It takes only two particles. Each particle has three position components (x, y, z) and three velocity components (v_x, v_y, v_z). With two particles, that's 12 degrees of freedom.

Monte Carlo methods are still slow, though. (Actually, the problems to which we apply Monte Carlo methods are slow.) It's definitely worthwhile to carefully consider any symmetries in the problem you're facing and use them to your advantage. For example, if the shape you are integrating is symmetric about some axis, find the area on one side only, then double the result. Make sure your known area is not bigger than it needs to be, and by all means devise your test for whether the points are within the area or not to be as efficient as possible.

Example 6.1.1
Use Monte Carlo integration to find the volume of a hemisphere of radius 1.

We can use the symmetry of the problem to find the volume of one quarter (the positive $\{x, y, z\}$ quarter) of the hemisphere, then multiply by 4 to get our final answer. For the known volume surrounding our unknown piece, let's use the unit cube: x, y, and z range from 0–1. This range of values corresponds exactly to the evenly-distributed random numbers given by random.random(). Here's code to do the job:

```
from random import random
from math import sqrt

# Since the radius is 1, we can leave it out of most calculations.

N = 1000000                    # number of random points in unit cube
count = 0                      # number of points within sphere
for j in range(N):
    point = (random(), random(), random())
    if point[0]*point[0] + point[1]*point[1] + point[2]*point[2] < 1:
        count = count+1

Answer = float(count)/float(N)
# Make sure to use float, otherwise the answer comes out zero!
# Also note that in this case the volume of our "known" volume (the unit
# cube) is 1 so multiplying by that volume was easy.

Answer = Answer * 4            # Actual volume is 4x our test volume.
```

```
print '''The volume of a hemisphere of radius 1
is %0.4f +/- %0.4f.''' % (Answer, 4*sqrt(N)/float(N))
```

The program gives a volume of 2.095 ± 0.004, which is consistent with the known value $\frac{2}{3}\pi r^3$.

6.2 Problems

6-0 Calculate $\int_0^\pi \sin(x)\,dx$ using Monte Carlo techniques. Report your uncertainty in the result, and compare with the known result.

6-1 Find the volume of the intersection of a sphere and a cylinder, using Monte Carlo techniques. The sphere has radius 1 and is centered at the origin. The cylinder has radius 1/2, and its axis is perpendicular to the x axis and goes through the point ($\frac{1}{2}$, 0, 0). (See figure 6.2.) Report your uncertainty, also.

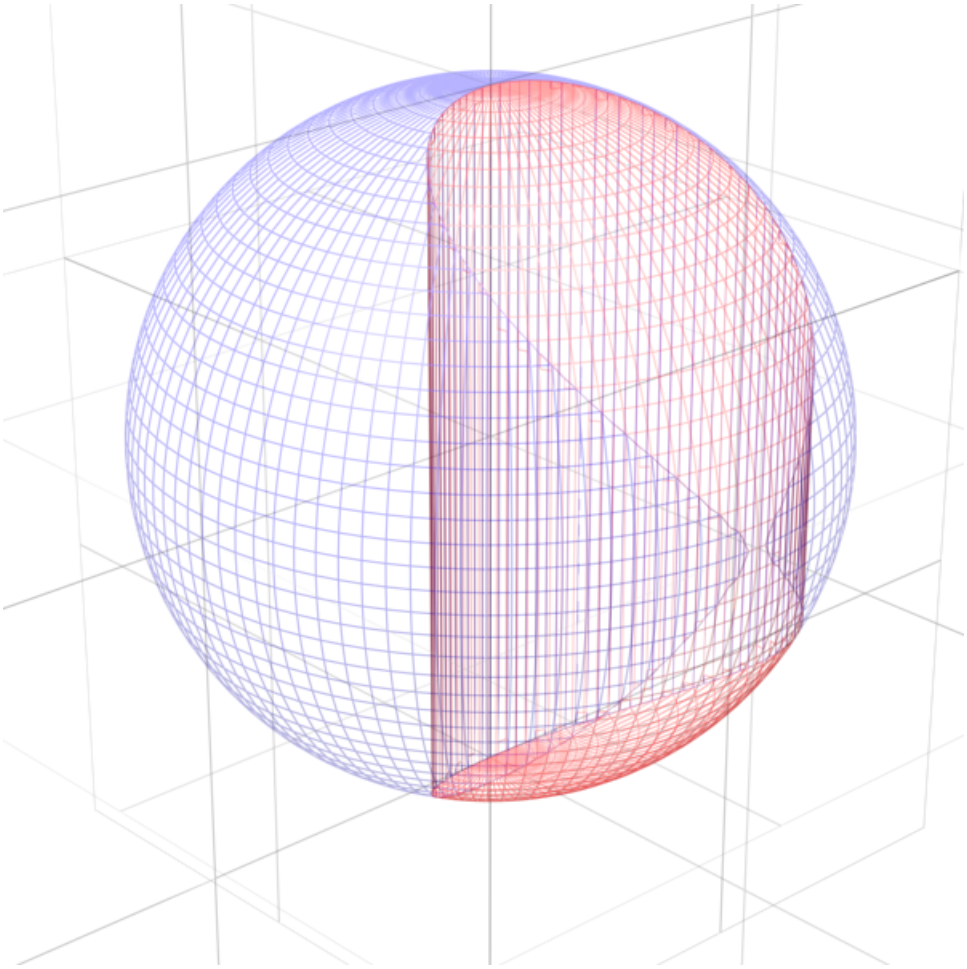


Figure 6.2: Figure for problem 1

6-2 The "volume" of a 2-sphere $x^2 + y^2 = r^2$ (AKA a "circle") is $(1)\pi r^2$. The volume of a 3-sphere $x^2 + y^2 + z^2 = r^2$ is $\left(\frac{4}{3}\right)\pi r^3$. The equation for a 4-sphere is $x^2+y^2+z^2+w^2 = r^2$. We can guess, by extrapolation from the 2-dimensional and 3-dimensional cases, that the "volume" of a 4-sphere is $\alpha\pi r^4$. Use Monte Carlo integration to estimate α.

Chapter 7

Stochastic Methods

Take a still beaker of water, and drip one drop of dye into the center. Over time the dye spreads out, and eventually distributes itself evenly throughout the water. How would we model this behavior computationally?

The direct method would be to calculate $\vec{x}$ and $\vec{p}$ for each molecule in the beaker, but this poses some computational difficulty. Assuming 10^{23} molecules, with six degrees of freedom each, and a 4-byte floating-point number for each degree of freedom, this direct method would require 2.4×10^{12} Tb just to store the state of the system at any instant.[1] In addition to the storage problem, the direct method has limited usefulness even if it could be implemented: We could calculate the time it would take for the dye molecules to evenly spread throughout the water, but then if we wanted to know how long it would take to distribute through twice the volume we'd have to do it again! The direct method doesn't give us any *understanding* of what's going on.

Instead of the direct method, we use *stochastic* methods. The fundamental idea behind these methods is that *large ensembles act in an "average" way even if individual elements are random.* We lose the details about each molecule, and only learn the behavior of the ensemble.

7.0 The Random Walk

We'll start with the simplest method, the random walk in one dimension. Start with a drunken frat boy standing on the sidewalk outside of the Bear at about 2AM on a Saturday morning. He can take a step east, or a step

[1] The "volume" of the internet was estimated to be merely 7.5×10^3 Tb in the fall of 2006.

west. For the sake of this model, we'll assume that the probability of either direction is 0.5, and the step lengths are all the same. The question we'd like to answer is, "how fast, on average, does this random walk move the frat boy away from the Bear?"

Here's a program that models his motion:

```
#!/usr/bin/env python

"""
Fratboy.py
Program to model 1-D random walk
"""

from random import choice
from pylab import *

N = 200       # number of steps

# set up storage space
x = zeros([N])
t = range(N)

# Do the walk
for i in range(1,N):
    if choice(['forward', 'back']) == 'back':
        # take a step back
        x[i] = x[i-1] - 1
    else:
        # take a step forward
        x[i] = x[i-1] + 1

RMS = array([sqrt(i*i) for i in x])

plot(t,x,'b-')
plot(t,RMS, 'g-')

show()
```

This program only tells us *one* walk, though. We need to know the *average* behavior of a drunk frat boy. To calculate this average behavior, we take several thousand non-interacting drunk frat boys, or —since non-interacting drunk frat boys don't exist[2] — we run the simulation several thousand times

[2] As has been determined experimentally, drunk frat boys in groups tend to all move in the same general direction, and burn furniture in the middle of the street. While random

to determine the average RMS displacement. Here's a program that does just that:

```
#!/usr/bin/env python

"""
fratboy_average.py
Program to model AVERAGE 1-D random walk
"""

from random import choice
from pylab import *
from scipy.optimize import curve_fit

def power(x,a,b):
    return a*x**b

steps = 200      # number of steps
boys = 2000      # number of fratboys

# set up storage space
x = zeros([steps])
t = range(steps)
x_sum = zeros([steps])
x2_sum = zeros([steps])

# Do the walks
for j in range(boys):
    for i in range(1,steps):
        if choice([0,1]):
            x[i] = x[i-1] - 1
        else:
            x[i] = x[i-1] + 1
    # add x, x^2 to running sums
    for i in range(steps):
        x_sum[i] = x_sum[i] + x[i]
        x2_sum[i] = x2_sum[i] + x[i]**2

# rescale averages
x_avg = [float(i)/float(boys) for i in x_sum]
RMS = [sqrt(float(i)/float(boys)) for i in x2_sum]

xlabel("Time (Step number)")
```

in some sense, this is not the random behavior we're interested in studying in a physics course.

```
ylabel("Average and RMS position (Steps)")
plot(t,x_avg, 'b-')

plot(t,RMS, 'g-')


# Check least-squares fit, see what the power dependence is.
# I assume that the RMS displacement goes as D = A*t^B
popt, pcov = curve_fit(power, t, RMS)

print "Power fit: y(t) = A * t^B:"
print "A = %f +/- %f." % (popt[0], sqrt(pcov[0,0]))
print "B = %f +/- %f." % (popt[1], sqrt(pcov[1,1]))

# Plot the curve fit on top of that last graph
plot(t, power(t, popt[0], popt[1]))
show()
```

An RMS displacement graph for this program is shown in figure 7.0. The RMS displacement goes as the square root of the number of steps. There's a good mathematical reason for this: The displacement after n steps is given by

$$x_n = \sum_{i=1}^{n} s_i$$

where s_i is the step length for step i, which in this case could be either ± 1. The square of the displacement is given by

$$x_n^2 = \left(\sum_{i=1}^{n} s_i \right) \left(\sum_{j=1}^{n} s_j \right) = \sum_{i=1}^{n} s_i \sum_{j=1}^{n} s_j \ .$$

We can break the double sum into two portions:

$$x_n^2 = \sum_{i=j} s_i s_j + \sum_{i \neq j} s_i s_j \ .$$

The second term ($i \neq j$) averages to zero, since the direction of the step is random. We're left with just the first term, which simplifies nicely to

$$\left\langle x_n^2 \right\rangle = \sum_{i=1}^{n} s_i^2 = n \left\langle s^2 \right\rangle \ .$$

In our present case $s_i = \pm 1$, so

$$\langle x_n \rangle = \sqrt{n}$$

as is consistent with the program output.

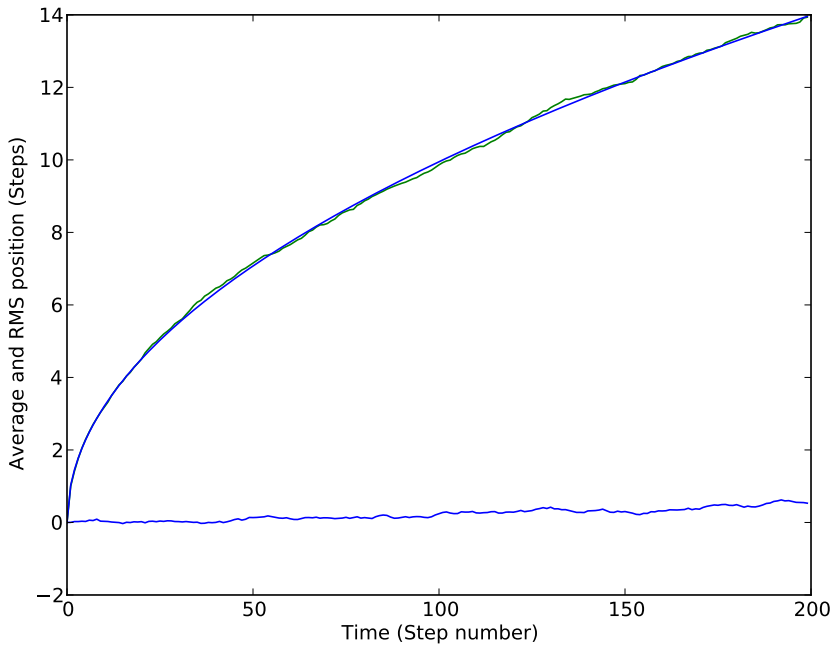


Figure 7.0: Average RMS Displacement for a one-dimensional random walk with a uniform stepsize of 1. This graph shows the average for 2000 walks. A least-squares fit is also shown for $x(t) = At^B$, where $A = 0.994 \pm 0.002$ and $B = 0.4991 \pm 0.0005$.

7.1 Diffusion and Entropy

We can use the random walk to model the diffusion of the drop of dye in water from the beginning of this chapter. For ease of display, the following program does only 2-D diffusion rather than 3-D, but the concept is the same.

```
#!/usr/bin/env python
"""
diffusion.py

Random-walk model of diffusion in a 2-d environment.
Starts with 400 particles in a square grid centered at (100,100).
At each step, the program picks each particle and moves it (or not)
```

```
one integer step in the x and y directions. If the move would
take the particle beyond the boundary of space (200x200), then
the particle bounces off the wall and moves the other direction.

The program plots the positions of all particles after each step.

Call the program with one argument: the number of steps to take.
"""

import sys
from pylab import *
from random import randint  # randint(a,b) picks a random integer in
                           # the range (a,b), inclusive.

# Allow animation
ion()

# set up graph window
figure(figsize=(10,10))

# Define droplet coordinates (all droplets) to be at point 100,100.
atoms = ones([400,2])*100

# show initial configuration
line, = plot(atoms[:,0], atoms[:,1], 'ro')
xlim(0,200)
ylim(0,200)
draw()
wait = raw_input("Press return to continue")

# How many steps to take?
N = int(sys.argv[1])

for i in range(N):
    # Go through all atoms
    for j in range(400):

        # Move each atom (or not) in the x and/or y direction.
        atoms[j,0] += randint(-1,1)
        atoms[j,1] += randint(-1,1)

        # Check for boundary collision
        x,y = (atoms[j,0], atoms[j,1])
        if x == 200:
            atoms[j,0] = 198
```

```
            elif x == 0:
                atoms[j,0] = 2
            if y == 200:
                atoms[j,1] = 198
            elif y == 0:
                atoms[j,1] = 2

        # See how things look now.
        line.set_xdata(atoms[:,0])
        line.set_ydata(atoms[:,1])
        draw()

wait = raw_input("Press return to exit")
```

It takes awhile (4-5 thousand iterations), but the "dye molecules" in this simulation eventually spread evenly throughout the container. (The simulation runs much faster if you don't replot at every step, of course!)

One characteristic of the simulation is that the system becomes more disordered with time. At the beginning, all of the molecules are on a single point at the center of the simulation, and at the end they're scattered about like legos on my son's bedroom floor after a hard day of playing. We quantify this "amount of disorder" as *entropy*.

In previous physics classes, you probably dealt only with *change* in entropy,

$$\Delta S = \frac{\Delta Q}{T} \, .$$

For a system with discrete states such as this diffusion simulation, it's possible to define the entropy exactly

$$S = -k_B \sum_i P_i \ln P_i \tag{7.1}$$

where P_i is the probality of finding a particle in state i. We can divide the simulation up into smaller segments (see figure 7.1) to estimate the values P_i.

Each grid square in figure 7.1 represents one state i. The probability (or relative frequency) of particles in each state is the number of particles per square, divided by the total number of particles. For example, in the final state (on the right in figure 7.1) the value of P_i for the top left state is 5/400, since there are 5 particles in that particular box and 400 particles total.

The entropy of the initial state is zero for this particular choice of states. The relative frequency for each state is zero for all but the center state, since

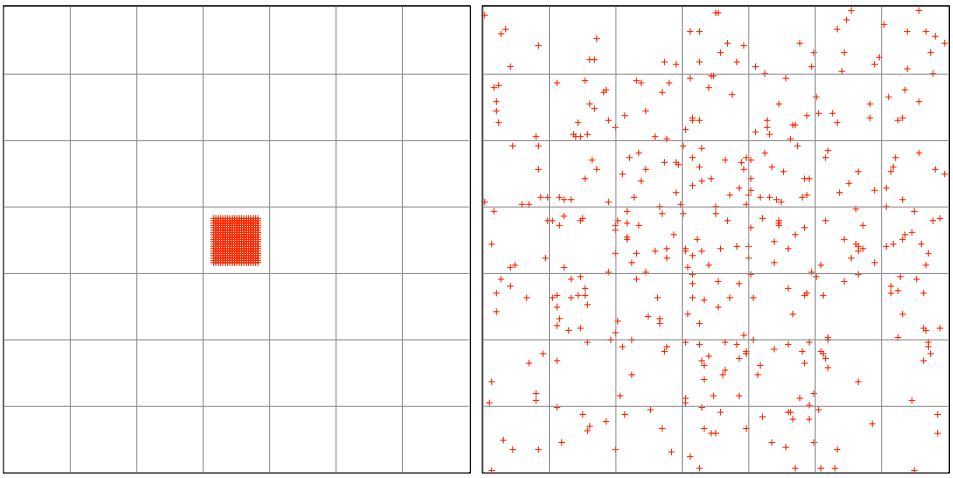


Figure 7.1: Output of the program diffusion.py, initially and after 5,000 steps. The light grid overlaid on both graphs is for calculation of entropy by equation 7.1. In the initial state $S = 0$, and in the final $S = 195$.

they're all empty. So $P_i = 0$ in equation 7.1 for all but the center square. In the center square, $P_i = 1$ since all of the particles are there, and $\ln 1 = 0$ so the total entropy is $S = 0$.

The entropy of the final state is calculated in the same way, but of course the number of particles in each box is not zero. Writing a program to do this is left as an exercise, in problem 2.

7.2 Problems

7-0 We showed in section 7.0 that the RMS displacement of the drunk fratboy goes as $\sqrt{\text{time}}$. That was done for a one-dimensional walk, though. Diffusion of dye molecule in a liquid would be a three-dimensional random walk, instead of one-dimensional. What is the time dependence of the RMS displacement for a 3-D random walk?

7-1 What happens in the one-dimensional random walk if the stepsize is also random? Model this situation where instead of randomizing the direction of each step, you pick a step with a random length on the range [-1, 1]. What is the time dependence of the RMS displacement in this case?

7-2 Modify the program diffusion.py (in the /export/312/shared/ directory) so that it makes a plot of entropy vs. time.

Chapter 8

Partial Differential Equations

8.0 Laplace's Equation

Knowing the electric potential in an area of space allows us to calculate the electric field in that area, and electric field tells us what we need to know about forces on charged particles in that area.

If we know where all the charges are in an area, we can calculate the potential from

$$V(\vec{r}) = k \sum_{i=1}^{n} \frac{q_i}{\vec{r_i} - \vec{r}} \,. \tag{8.1}$$

The problem with equation 8.1 is that we rarely know the exact charge distribution in an area. We're far more likely to know the *potentials* at certain points in the area, which is quite different from knowing the *charges* in that area.

Laplace's equation describes the potential in a charge-free region. That "charge-free" region can include all the areas between the charges in any area we want, so we can use it to get beyond the problems in equation 8.1. The derivation of Laplace's equation is good material for a different course than this one, so let's just start with the equation itself:

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = 0 \tag{8.2}$$

We'll start in the usual way, by breaking our space into a discrete grid of points. Then we'll develop a simplified expression of Laplace's equation, and use that to solve for the potential V.

The simplest way of calculating the derivative uses equation 2.13:

$$\frac{\partial V}{\partial x} \approx \frac{V(x_{i+1}) - V(x_i)}{\Delta x} \tag{8.3}$$

Let's simplify our notation a bit before we go further: we'll define $V_i \equiv V(x_i)$, and so on. So $V_{i+1,j,k}$ would refer to the potential at the point (x_{i+1}, y_j, z_k). Getting back to equation 8.3, this is actually the derivative at the point halfway between x_i and x_{i+1}. Let's call that point $x_{i+\frac{1}{2}}$. We could also use equation 8.3 to find $\frac{\partial V}{\partial x}$ at $x_{i-\frac{1}{2}}$:

$$\frac{\partial V}{\partial x} \approx \frac{V_i - V_{i-1}}{\Delta x} \tag{8.4}$$

Of course, for equation 8.2 we actually need the second derivative. We need this second derivative at x_i, so we use the difference in the first derivative at $x_{i+\frac{1}{2}}$ and $x_{i-\frac{1}{2}}$ in equation 2.13.

$$\frac{\partial^2 V}{\partial x^2} \approx \frac{V_{i+\frac{1}{2}} - V_{i-\frac{1}{2}}}{\Delta x^2} = \frac{V_{i+1} + V_{i-1} - 2V_i}{\Delta x^2} \tag{8.5}$$

As it turns out, equation 8.5 is better than we deserve from the method we used to calculate it. It's exactly the same as equation 2.21, which is good to third order in Δx. It's also better than we *need*, since we're never going to actually calculate that derivative!

Let's go back to equation 8.2 and put in our results from equation 8.5. In three dimensions, we get

$$V_{i,j,k} \approx \frac{1}{6}\left[V_{i+1,j,k} + V_{i-1,j,k} + V_{i,j+1,k} + V_{i,j-1,k} + V_{i,j,k+1} + V_{i,j,k-1}\right] \tag{8.6}$$

And here's the good part: equation 8.6 tells us that the solution to Laplace's equation is such that V at each point is just the *average* of the potential at the surrounding points. This has one very important implication in electrostatics: it means that there are no local minima or maxima of V in charge-free regions.[1]

Relaxation Method

The "average" nature of the solution to equation 8.2 suggests the *relaxation method* as a way of solving for the potential in charge-free regions.

[1] For further discussion of this marvelous result, see [5].

1 Populate your solution grid with some initial guess of what the solution should look like. Anything works — zero is fine — but if you have even a coarse estimate of the solution you can save quite a bit of calculation time by starting with that.

2 Go through your grid of (x, y, z) points, recalculating each point to be the average of the surrounding points.

3 As you go, keep track of how much things change. When the largest change is less than your solution tolerance, stop.

Of course there are some subtle —or not-so-subtle— points to keep track of when using the relaxation method. First, you need to make sure that the area of space in which your calculations take place is actually *bounded*. In other words, make sure that the boundary of the area you're relaxing is set to some fixed potential. Zero usually works well, and the easiest way of ensuring this is to initially set your guess to something that has zero on all the boundaries, then not re-calculating the boundary elements.

Second, your solutions are going to be very boring unless there are some fixed potential points inside the area. Don't recalculate the potential at those points! They're supposed to be fixed.

A plot of the potential near a "parallel-plate capacitor" on a 100×100 grid is shown in figure 8.0. Note that this is the solution for V in a 2-D region: in 3-D this is the equivalent to the potental around two infinitely long parallel plates.

Over-Relaxation

We can speed the relaxation method considerably by using "over-relaxation"[4]. At each point on the grid, calculate the new value of V as before, but instead of assigning the new value to the grid point calculate the *change* in V: $\Delta V = V_{new} - V_{old}$. Multiply this ΔV by some factor α, and add the result to the original to get the new value of V at that point. In equation form,

$$V_{new} = \alpha \Delta V + V_{old} \tag{8.7}$$

The reason this method is faster is that V moves farther each step. It may even move past the "correct" value, but then it will just move back the next step. Overall, the solution converges faster.

The factor α should be between 1 and 2. $\alpha = 1$ corresponds to the regular relaxation algorithm, so there's no benefit. Values of α greater than 2 cause the algorithm to become unstable: V will oscillate from step to step

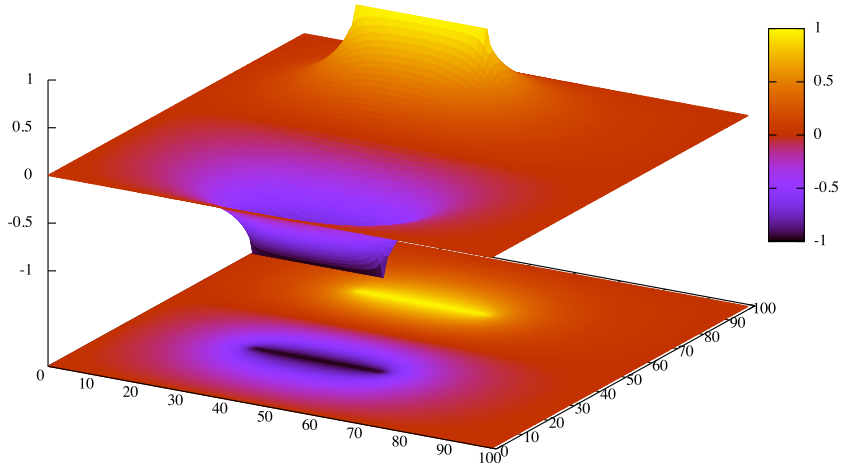


Figure 8.0: 2-D relaxation solution for V in the region surrounding two parallel plates held at opposite potentials. Both top and bottom show the same information in different ways.

without converging on the solution. The optimal value of α depends on the details of the problem, but values between 1.2 and 1.5 seem to work well most of the time.

8.1 Wave Equation

For our next partial differential equation, let's consider the wave equation,

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2} \ . \tag{8.8}$$

We can derive this equation to some degree of plausibility by starting with a string, as shown in figure 8.1. As you might expect by now, we break the string into segments of length Δx and label each segment with an index i. The mass per length of the string is μ, so the mass of each string segment

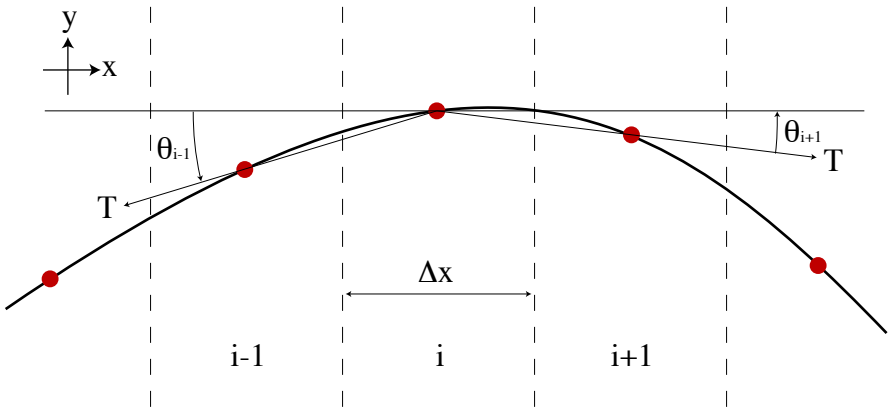


Figure 8.1: A bit of string for deriving the wave equation. The vertical scale is greatly exaggerated for clarity.

is $\mu\Delta x$. Next, bring in Newton's second law, $F = ma$, to find the vertical acceleration $\frac{d^2y}{dt^2}$ of segment i. The force on the string segment is just the sum of the vertical components of tension on each side, since the horizontal components of T must necessarily cancel out. Taking the angles θ to be positive in the directions shown, Newton's second law becomes

$$\mu\Delta x\frac{d^2y}{dt^2} = T\sin\theta_{i+1} - T\sin\theta_{i-1}\ . \tag{8.9}$$

For small angles, the distance between points is approximately Δx, so

$$\sin\theta_{i+1} \approx \frac{y_{i+1} - y_i}{\Delta x} \tag{8.10}$$

and

$$\sin\theta_{i-1} \approx \frac{y_i - y_{i-1}}{\Delta x}\ . \tag{8.11}$$

Substituting equations 8.10 and 8.11 into equation 8.9 and rearranging slightly, we obtain

$$\frac{d^2y}{dt^2} \approx \left(\frac{T}{\mu}\right)\frac{y_{i+1} + y_{i-1} - 2y_i}{\Delta x^2}\ . \tag{8.12}$$

But the right-hand side of equation 8.12 is just the second derivative of y with respect to x (See equation 8.5), so we now have the discrete-particle equivalent of equation 8.8, with $c = \sqrt{\frac{T}{\mu}}$.

For a numeric solution to the wave equation, we apply our numeric second derivative equation to each side of equation 8.8. Since we're doing second derivatives with respect to both time and space, we need to make sure the notation is clear. We'll use i to indicate steps in space, and n to indicate steps in time, so

$$y_{i,n} \equiv y(i\Delta x, n\Delta t)$$

and then equation 8.8 becomes

$$\frac{y_{i,n+1} + y_{i,n-1} - 2y_{i,n}}{\Delta t^2} = c^2 \left[\frac{y_{i+1,n} + y_{i-1,n} - 2y_{i,n}}{\Delta x^2} \right] . \tag{8.13}$$

Solve equation 8.13 for $y_{i,n+1}$ to obtain

$$y_{i,n+1} = 2\left[1 - r^2\right] y_{i,n} - y_{i,n-1} + r^2 \left[y_{i+1,n} + y_{i-1,n}\right] \tag{8.14}$$

where $r \equiv \frac{c\Delta t}{\Delta x}$.

Let's stop and think what equation 8.14 tells us. Remember that i refers to position and n refers to time; so it's saying that the position of the string for *this* location at the *next* time step ($y_{i,n+1}$) depends on where the string segment is now ($y_{i,n}$), where the segment used to be ($y_{i,n-1}$), and where the segments on each side are at present ($y_{i+1,n}, y_{i-1,n}$).

To use equation 8.14, we need two starting configurations. We need to know the y value of the string at each i for n, and the y value for each i one time-step in the past, at $n-1$. The easiest way of doing this is to start with the string at some non-equilibrium position, and not moving. Make two arrays of y values: y_{old} and y_{now}. Since the string is not moving, their values should be the same. This produces a wave in a manner equivalent to plucking a guitar string: the string is pulled to one side, held briefly, and released.

Another way of generating the required two starting configurations is to create two slightly different arrays of y values. A simple example would be to create y_{old} with an equilibrium configuration (zero throughout) and then create y_{new} to be also zero but with one element just slightly positive. This produces a wave in a manner similar to striking a piano string with a felt hammer.

8.2 Schrödinger's Equation

Quantum theory is one area that seems to require computers for further progress. The time-independent Schrödinger's equation,

$$-\frac{\hbar^2}{2m}\nabla^2\psi + V(\mathbf{r})\psi = -i\hbar\frac{\partial}{\partial t}\psi \tag{8.15}$$

describes the wavefunction ψ for a quantum particle. This wavefunction can then be used to calculate the probability of finding the particle in a given location as well as the position, velocity, momentum, and so on.[2] We can solve Schrödinger's equation analytically for simple cases: particles in boxes, harmonic oscillators, and hydrogen atoms are examples. But beyond that, we're pretty much stuck.

We're not going to get unstuck in part of one chapter of a book on computational physics, either! This is a complicated topic. But we can get a short survey of one useful technique if we limit ourselves to the time-independent Schrödinger's equation, and to bound particles in one dimension.

$$-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}\psi + V(x)\psi = E\psi \tag{8.16}$$

This looks straightforward enough. $V(r)$ is the potential well in which the particle is bound, $\hbar$ and m are known, and E is just some constant. There are conditions on ψ, though: ψ must be continuous, and if V is not infinite then the derivative of ψ must also be continuous. There is also a normalization condition: the probability of finding the particle *somewhere* is exactly one, so

$$\int_{-\,\mathrm{inf}}^{\mathrm{inf}} \psi * \psi \; dx = 1 \tag{8.17}$$

The fascinating thing about solutions to this equation — the *quantum* thing, if you will — is that these conditions result in valid solutions only for certain values of E. The usual way of showing this is to work through an actual solution[12] but this is a computational class, so let's just take a brute-force approach and see what happens.

Imagine a square potential well, symmetric about the origin, with width $2L$. This well contains a single particle. Since the particle is bound by the potential, we know that the particle energy $E < V$ for $-L < x < L$. By symmetry, we can see that there are two possible classes of solution to Schrödinger's equation for this configuration: symmetric and anti-symmetric

[2] Take a Quantum class for more details, if you haven't already.

solutions. From the normalization condition, we can see that ψ must go to zero for large x. Two possible solutions that meet these criteria are shown in figure 8.2.

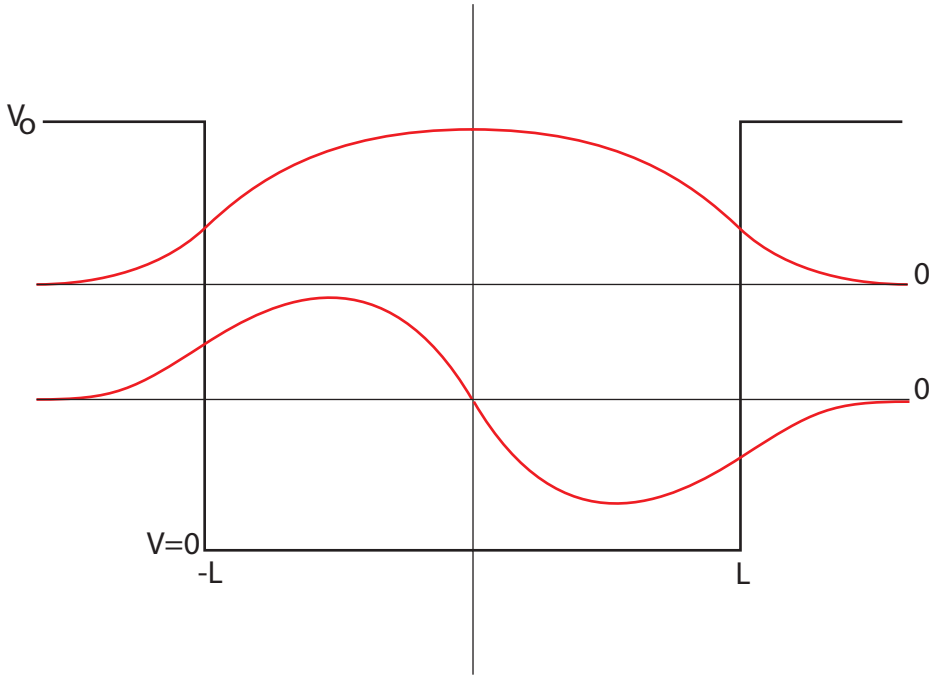


Figure 8.2: Two possible solutions to Schrödinger's equation, overlaid on a square potential well. The upper solution is symmetric, the lower is anti-symmetric. The two are on different axes: don't interpret the relative vertical positions of the graphs as meaning anything!

What we want to do is find the value(s) of E that give valid solutions to ψ. We'll do this by starting with a guess of the value of E. We'll take this guess and use our ODE-solving techniques to find ψ, starting from the center and working out to positive x values. We will know the solution is a good one if $\psi \to 0$ at large values of x, and we will adjust E until we find a good solution.

For initial conditions in the symmetric case, we will start with $\psi = 1$ and $\frac{d\psi}{dx} = 0$. This value of ψ is almost certainly incorrect, but we can scale

it to its correct value later by using equation 8.17. For the anti-symmetric case, the appropriate initial conditions are $\psi = 0$ and $\frac{d\psi}{dx} = 1$. Again, that value of $\frac{d\psi}{dx}$ is almost certainly incorrect, but we can normalize ψ later to correct it.

Equation 8.16 is a second-order ODE. We can simplify it further without loss of generality by redefining our system of units so that $\hbar = 1$ and $m = 1$:

$$\ddot{\psi} = 2 * (V(x) - E)\psi \tag{8.18}$$

For the finite square well in figure 8.2, it's easy to define the potential $V(x)$:

```
def V(x):
    """
    Potential in which the particle exists.
    L = 1
    """
    if x<1.0:
        return 0                # square well
    else:
        return Vo
```

We can then define the ODE for solution with odeint():

```
def SE(y,x):
    """
    Returns derivs for the 1-D TISE, for use in odeint.
    Requires global value E to be set elsewhere.
    Note that we are using x as time here... Python doesn't care!
    """
    g0 = y[1]
    g1 = 2.0*(V(x)-E)*y[0]
    return array([g0,g1])
```

And we can plot the resulting wavefunction ψ. If our value of E is too large or too small, ψ diverges quickly, as shown in figure 8.3. This method is called the "shooting method", for reasons which are fairly obvious.

We can automate the process of finding the correct value of E, of course. In section 2.0 we saw several ways of finding a zero of a function: this is no different except that the "function" for which we want a zero is just something that returns the last value of ψ in our odeint solution. Sample code for this, using the brentq() root-finding function, is shown below.

```
#!/usr/bin/env python

"""
SEsolve.py
```

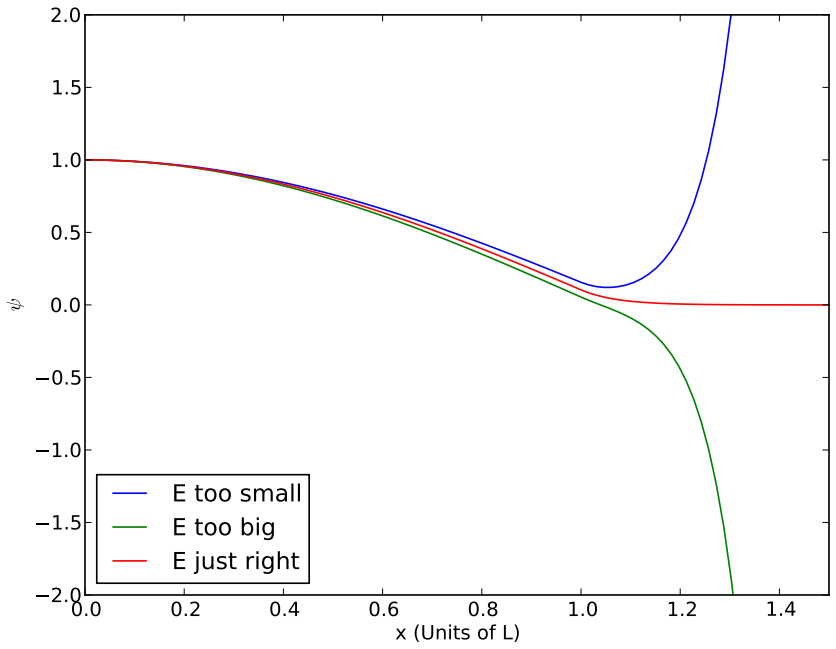


Figure 8.3: ψ diverges quickly if E is slightly off!

```
"""Uses shooting method to find even-parity solutions to the finite square well
Takes two guesses of  energy from the command line, finds a bracketed energy
eigenvalue E, and plots the corresponding wavefunction.

hbar, m, and L are all taken as 1.

The parameter b is the point in units of L at which to check if the
wavefunction diverges. b must be greater than 1, of course.

Solutions are not normalized.
"""

from pylab import *
from scipy.integrate import odeint
from scipy.optimize import brentq
import sys
```

```
b = 2.0
Vo = 20.0        # Potential outside square well
steps = 100
E = 0.0          # global variable, changed by Final_Value()

def V(x):
    """
    Potential in which the particle exists.
    L = 1
    """
    if x<1.0:
        return 0             # square well
    else:
        return Vo


def SE(y,x):
    """
    Returns derivs for the 1-D TISE, for use in odeint.
    Requires global value E to be set elsewhere.
    Note that we are using x as time here... Python doesn't care!
    """
    g0 = y[1]
    g1 = -2.0*(E-V(x))*y[0]
    return array([g0,g1])

def Final_Value(energy):
    """
    Calculates psi for this value of E, and
    returns the value of psi at point b to check divergence.
    """
    global y
    global E
    E = energy
    y = odeint(SE,yo,x)
    return y[-1, 0]              # return final value (psi at b)

y = zeros([steps,2])
yo = array([1.0, 0.0])           # initial psi and psi-dot.
x = linspace(0, b, steps)
E1 = float(sys.argv[1])
E2 = float(sys.argv[2])

answer = brentq(Final_Value, E1, E2) # use brentq to solve for E-> psi=0 at b.
```

```
print 'Eigenvalue found at E = %.8f' % answer

plot(x, y[:,0] )
xlabel("Position (Units of L)")
show()
```

It is a simple matter to change this code to find the anti-symmetric solutions, and one can find different solutions by giving the program different search limits E_1 and E_2. One can also easily change the shape of the potential well from square to just about anything else. With some creativity, it is also possible to adapt this method to one-sided wells and other asymmetric cases.

8.3 Problems

8-0 Write a program that uses the relaxation method to calculate and plot the potential in a 2-D region bounded by zero potential. There are three files in /export/250/shared/ that describe a grid of fixed potentials: dipole.txt, plates.txt, and cage.txt. These files contain "0" in the non-fixed-potential regions, and ± some value on grid points at which the potential is held fixed. Make sure your program gives sensible results for dipole.txt and plates.txt. (The solution to plates.txt is shown in figure 8.0.) Save an image file of a plot of your solution to cage.txt, and turn it in with your program.

8-1 Find all bound energy levels for a particle in a triangular well. The potential should be $V = 0$ everywhere except in the well, in which it drops linearly from zero at $x = -L$ to $V = -50$ at $x = 0$ and then climbs linearly to $V = 0$ at $x = L$. Use the simplified unit system from this chapter: $\hbar = 0$, etc.

Appendix A

Linux

Linux is a free, open-source, community-developed computer operating system. Open-source means that the actual code is available to anyone: you can take it apart, see how it works, and modify it if you desire. It's community-developed, meaning that the people who develop it are the ones who use it. As a result, it tends to lean towards functionality over fashion, and speed over eye-candy. The "free" bit is self-explanatory: you are welcome to pay for Linux, but you can also download it without charge.

Linux is functionally equivalent to Unix, which has been the premier scientific computing platform since long before the development of the Macintosh or Microsoft Windows. There is an enormous quantity of scientific software available under Unix, and it's a good idea for anyone going into a scientific field to have at least a passing familiarity with it.

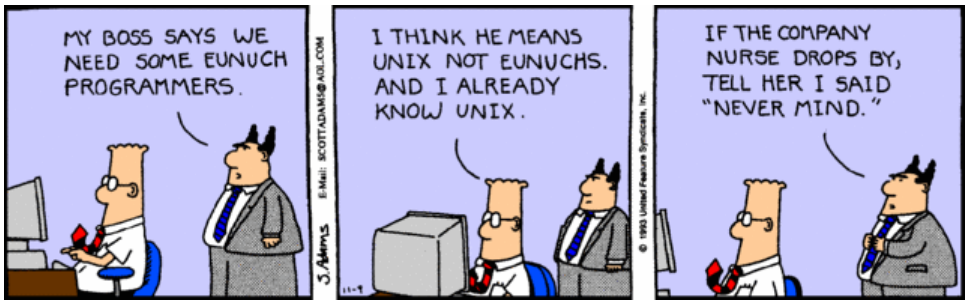


Figure A.0: One good reason to know Unix.

A.0 User Interfaces

Since 1984, computer operating systems have moved almost entirely to the Graphical User Interface (GUI) from the original "Command-Line". This change has made computers much more accessible to the average user. GUIs are generally intuitive, easy to learn, and very user-friendly.

There is a two-fold disadvantage of the GUI over the command-line, though. First, the GUI lacks power compared to the command line. If you had several hundred text files, and needed to change every occurrence of the word "dog" to "cat" in those files, you'd be in for a day's work in the GUI. The same job could be done with one line[1] in a command-line interface. It is also very easy to string command-line programs together, so that the output of one becomes the input of the next. This allows a set of small single-purpose programs to be used in powerful ways.

Second, the GUI is much more difficult to program. If you want to take a file of numbers representing, say, the position of the laser spot for an experiment to measure the value of G, find the local maxima of the data, and write out another file with maxima and time data, it's relatively easy to write a quick-and-dirty program that will work with a command-line interface. The amount of extra programming required to make something like this work with a GUI is inconvenient.

A command-line interface is definitely harder to learn, though.

A.1 Linux Basics

The Operating System

It's important to understand the distinction between the operating system and everything else, including the command line. The operating system (OS) is what starts while your computer is booting. It is in charge of operating all your computer's hardware, talking to the network, and running programs such as your web browser, text editor, GUI, and your "shell". It also manages your "file system", which is the arrangement of everything on your hard drive including folders, files, programs, and security privileges.

The File System

Files in a Linux system are arranged in a *directory tree*. The root of this tree is `/`, and all other directories are attached to `/` in some way. For

[1] `perl -pi -e 's/ dog / cat /g' *.txt`, see [15].

example, when you log into one of the Linux machines in this lab, you will start in your "home directory", `/export/home/<username>/`. This means you are in a directory namaed with your username, which is contained in the directory `home`, which is contained within `export`, which is contained within `/`. You might have other directories within your home directory, such as `data`, `papers`, `homework`, and so on. Each of these subdirectories can contain more subdirectories or files.

Directories within Linux are identical to "Folders" in Windows or Mac OS X. Each file in a directory must have a unique name, of course. Files within *different* directories may have identical names, since their full names (including the entire directory path) would be different.

Each directory contains a minimum of two files: "." and "`..`". The first one is the current directory, and the second is the current directory's parent directory. So if you were in directory `/export/home/<username>/homework/`, and wanted to refer to the file data.txt in `/export/home/<username>/`, you could abbreviate it as `../data.txt`. Why it would be important to have a directory item (.) refer to the directory itself will become clear later... Another useful shortcut is the "`~/`" directory, which is shorthand for your home directory.

Most Linux filenames come in two parts: the name and the suffix. The file you are currently reading, for example, is named `linux.tex` on my computer. This file is a LaTeX file, and by convention they end in `.tex`. Similarly, Mathematica files are usually named something with `.nb` at the end, text files end in `.txt`, and so on. The suffix is not required for functionality in Linux, though: they are merely helpful for the user to figure out what they're supposed to be. Renaming `linux.tex` something like `Windows.exe` would not change the contents of the file, or how the OS treated it, or anything else.[2] It is not uncommon to see multiple suffixes on a file, such as `backup.tar.gz`, which would indicate an archive file (.tar) that has been compressed with gzip (.gz).

One thing to be cautious of when naming files: Linux doesn't particularly like spaces in filenames. Spaces are the delimiters between commands, files, and options in the command line. If you had a file named `homework seven`, and wanted to edit it, the command `edit homework seven` would send the operating system looking for files named `homework` and `seven` to edit. You can get around this problem by *escaping* the spaces — putting a \ character

[2]One minor change worth noting, though: changing the suffix will change the appearance of the file's icon if you're using a GUI. It's recommended that you use a file suffix that matches the file type for this reason.

in front of the space to indicate that it is a *space* rather than a break between separate filenames. `edit homework\ seven` would do what you want, there. The best policy, though, is to avoid spaces in filenames.

A.2 The Shell

The shell —your command-line interface— is a program just like any other. What it does is take commands from the user and ask the OS to perform the corresponding actions. These actions might include running other programs, moving files around, etc.

Most shell commands are actually small programs, and as such they have various options and optional targets. Options are usually indicated by `-`, and targets are usually files listed at the end of the command. An example would be `ls -l homework`, which would list (ls) in long form (-l) the files in directory `homework`.

The default shell on most Linux systems is the bash shell[3]. Here are some useful shell commands:

man Probably the most important command if you're new to the command line: it prints out the MANual pages for the target. Try `man ls`.

ls LiSt files. Important options include -a for "All" files, and -l for a long listing that includes size, permissions, owner/group information, and so on. If no target is given, ls gives the contents of the current directory.

cp CoPy files. `cp original.txt copy.txt` would make an exact copy of the file "original.txt", named "copy.txt". This can also be used across the directory structure: `cp original.txt /some/other/directory/` will put a copy of original.txt in /some/other/directory/. One particularly useful option is -R, which Recursively copies entire directories.

mv MoVe files. This works exactly like copy, except the original will be removed.

rm ReMove files. This deletes files permanently. (Actually, it "unlinks" the file by removing the information about where the file is on the drive from the directory tree and marking that area of the drive as

[3]There was an earlier shell called the "Bourne Shell", named after its programmer. The bash shell was written as a replacement for the Bourne Shell, thus the name: "Bourne Again SHell". Most Unix programmers have a weird sense of humor, even by physicists' standards. See "less", page 160.

available for use. The information is still there, until those drive sectors are overwritten.) Be particularly careful with the -r option, which removes files or directories recursively. `rm -r *` can be spectacularly devastating — see the section on wildcards, page 160.

cd Change Directory. This command allows you to move around the directory tree. Assume you had three folders in your home directory: homework, data, and papers. From your home directory, you could give the command `cd homework` to move to the homework directory, then `cd set-5` to move to the subdirectory `set-5` within homework. Equivalently, you could do both at the same time with `cd homework/set-5`. If you then wanted to move to the `data` directory within your home directory, you could give the command `cd ~/data`, or `cd ../../data`, or `cd /export/home/<username>/data`. Note that the `..` directory is particularly useful, here: it is the *parent* directory of whatever directory you are currently in.

pwd Print Working Directory. This is useful when you forget where you are in the directory tree: it tells you your current path.

mkdir MaKe DIRectory. This command creates a folder with the given target name. For example, if you are in `~/labs/` and want to create a new folder for your next experimental results, `mkdir Nobel` will create an appropriately-named directory within the current directory. This can also be used with different pathnames, of course: if you want to make a directory within a different area on the tree, just give mkdir the path it needs. From `~/labs/nobel/`, for example, the command `mkdir ~/homework/set-8/` will create the directory `set-8` in your `homework` directory.

The -p option will cause mkdir to create all directories in the given path, not just the last one. `mkdir -p ~/this/is/a/long/path` will create —if they don't exist already— five nested directories within your home directory.

rmdir ReMove DIRectory. The directory must be empty.

cat conCATinate files. What this actually does is print the source file(s) out to the target file. if either source or target is missing, cat uses the keyboard and screen. So `cat file1 > file2` will list the contents of `file1` to the file `file2`, obliterating what was already in `file2` if anything was there. `cat file1` is missing a target, so it will dump the

contents of `file1` to the screen. `cat > file2` is missing the input file, so it will happily wait for keyboard input, and copy everything you type to `file2`. (Ctrl-d will signal end-of-file in this case.) You can probably figure out what `cat file1 file2 file3 > bigfile` would do.

less Cat is not the best program for looking at what's in a file, since everything other than the last page disappears off the top of your screen almost instantly. The program 'more' was written to deal with this: it would show you more of the file, one page at a time. Eventually, someone wrote an improved version of more, with lots of options and so on. This program was more than more, and since less is more —more or less— more more was named 'less'.

Anyway... `less bigfile` will show you the contents of bigfile, one page at a time.

head Shows the first 10 lines of a file.

tail Shows the last 10 lines of a file.

sort Sorts the lines of an input into alphanumeric order.

wc Word Count. Reports the number of lines, words, and characters in a file.

passwd Set your password. Due to some specifics of the file-sharing system in use in the lab, you will have to use `yppasswd` to change your password, though.

Wildcards

Wildcard characters are characters that can mean multiple things.

The most common wildcard character is `*`, which roughly translates, in the bash shell, as "Any number of any character". `ls *.tex` would list all LaTeX files in the current directory, but it would not list `texas` or `lab-manual.tex.gz`.

Another useful wildcard is `?`, which means "*one* of any character". While `ls ca*` would list files like `cat`, `cake`, `cab`, and `catastrophe`, `ls ca?` would list only `cat` and `cab`.

Example A.2.1
You need to copy all Mathematica notebooks in your `homework` directory to the directory `~/notebooks`, then delete the originals.

One solution: `mv ~/homework/*.nb ~/notebooks/`

Although not technically a wildcard, the 'tab' key is certainly one you should know. If you begin typing a command or filename and then press 'tab', the shell automatically completes the name for you if it can. If you had three files in a directory named `cat`, `catalogue`, and `catastrophe`, and you wanted to look at the contents of `catalogue`, you could type `less catal(tab)` and the shell would fill in the rest of the filename. If the shell can't figure out what you mean, it will beep at you when you press 'tab'. Pressing 'tab' again will cause the shell to list the options it thinks you may mean, so you can continue typing letters until you have a unique selection.

There are other wildcards also, and entire books on how to use them. [8]

Pipes

Each of the commands explained in this chapter is actually a program: A small, finely-crafted single-purpose program. The true power of Unix-y systems comes from the ability to string these small programs together. The output of one becomes the input of the next, and so on. A single command, then, can invoke several programs, each of which does its thing and passes the results to the next.

Two of the important commands regulating this behavior are "|" and ">". The first one, "|", is the "pipe" command. It simply means to take the output of whatever came before the pipe and give it as input to whatever comes after. The second, ">", means "send the output to ...". If you follow ">" with a filename, or something that the OS interprets as a file name, then the results will be written to that file.

Some others which are worth brief mention include ">>", which means "Append to ...", and "<", which translates roughly to "Get from ...".

Example A.2.2
You need a text file named "paper-length" that contains a list of all the LaTeX files in a directory and the number of words in each, sorted by size.

One solution: `wc -w *.tex | sort -n > paper-length`

A.3 File Ownership and Permissions

Every file on a Unix system is owned by one particular user and is associated with one particular group of users. There are specific *file permissions* associated with the user, group, and "other" for each file. These file permissions include "Read", "Write", and "eXecute". You can see what the permissions are by using the command `ls -l`. This "long" listing will show the permissions, owner, and group for each file along with file size and other information. Here's an example of a long file listing:

```
-rwxrw-r-- 1 eayars phys300B 355B Dec 19 21:35 XF07.py
```

This file has user "eayars" and group "phys300B". It is a regular file, 355 bytes long, was last modified at 9:35 pm on December 19, and is named `XF07.py`.

Those first ten characters deserve some more explanation. The first character indicates the file is a regular file (-). (The most common other character there (d) indicates that it's a directory.) The next nine characters (`rwx r-x r--`) indicate the file permissions for the user, group, and "other". The user permissions (the first three characters) indicate that the owner of the file has Read, Write, and eXecute privileges. This person can read it, change it, and actually run it if it's a program. The next three characters tell us that anyone in the group "phys300B" can Read the file and eXecute it, but can't Write on it. The last three characters indicate that everyone else can only Read it.

Directories are slightly different. An `x` in the directory's permissions indicate "access" permission. The `w` indicates write ability, and `r` indicates the ability to list the directory contents. It is possible to have access without read permission: in that case one could retrieve specific files but could not get a list of files available. So if a directory had permissions set to `rwxr-x-w-`, then the user could do anything, the group could list and access the contents of the directory, and everybody else could write to the directory but could never see what had been written.

Setting permissions is done with the `chmod` command, which CHanges MODes for a file. `chmod +x program.py` would add execute permissions to program.py for everyone. `chmod o-w program.py` would remove write access for the general population (o), leaving the write status unchanged for the user (u) and group (g). This allows you to set permissions for user, group, and other seperately. These options can be combined in many ways, and multiple options can be used, separated by commas.

It is also possible to change the group of a file that you own, by using the `chgrp` (CHange GRouP) command. You can only change the file's group to

a group for which you are a member.

Example A.3.1
You need a folder that can be used as a "drop box" for your fellow 300B students. In other words, a folder into which they can place files, but can't get them out or see what's there. People not in the phys300B group should have no permissions in this folder.

Solution:

```
mkdir dropbox
chgrp phys300B dropbox
chmod go-rx,g+w dropbox
```

A.4 The Linux GUI

The command-line interface may be the most powerful option for controlling a computer, but it is not the only option. Linux usually comes with the "X" GUI, a powerful multi-purpose windowing system which comes in many flavors such as Gnome, KDE, xfce, and many others. The computers in room 123 are configured with the xfce user interface by default. If you can run Windows or Macintosh, you will feel quite at home within any of these systems, although they lack some of the eye-candy and polish of the latest offerings from Microsoft or Apple. And of course, there are easy-to-use GUI-based programs to do just about anything you need.

A.5 Remote Connection

When you log in to your account in room 123, you are actually accessing files stored on the Physics department's Linux server, "richard". The processor, memory, screen, and most of the hard drive details are handled by your local machine, but anything on `/export` or its subdirectories is actually stored on the server. This means that you can log into any of the machines in room 123 and get the same user settings and files. It also means that you can access those same files from any computer with an internet connection.

The Linux server allows connection via `ssh`[4], a method of generating secure encrypted communications between two computers.

[4]Secure SHell

Connection from Unix machines (including Macintosh)

To connect from a unix-based machine (Linux or Macintosh) use the terminal and give the command `ssh username@richard.csuchico.edu`. After giving your password, your terminal window will become a terminal window for the server. You can use any command here that you could use from a terminal window in room 123.

To copy files from the server, use the command `scp`[5], as in
`scp username@richard.csuchico.edu:remote-file local-destination`.
For copies to the server, reverse the two file designations:
`scp local-file username@richard.csuchico.edu:remote-destination`.
There are also various graphical tools to help you with this. On the Macintosh, the program "Fetch" works well, and is freely available to students and faculty. Under Linux, use the built-in "connect to server" feature under the "places" menu on Ubuntu Linux, or in some equivalent location on other flavors of Linux. (Be sure to select "ssh" as the service type.)

One option for connecting remotely from unix machines is "sshfs"[6] which makes your computer believe that the remote files are part of your local file system. Setting this up on a Macintosh is trivially easy, and only slightly less so on Linux: Google it for up-to-date information as sshfs is changing rapidly as of this writing.

Connection from Windows machines

For a shell connection to the Linux server from a Windows machine, download and install the program "PuTTy". The configuration is straightforward: connect to richard.csuchico.edu with your username, and give your password when prompted. The terminal window that opens will allow you to do anything that you could do from within a terminal window in room 123.

To copy files between a Windows machine and the Linux server, download and install the program "WinSCP", which gives you a user-friendly drag-and-drop interface between the two machines.

One thing you must be aware of is that Windows and Unix machines use different methods of marking line endings in text files. Programs created on a Windows machine will not run correctly on a Linux machine unless the line-endings are converted. Similarly, files created on a Linux machine will appear as a single line with some extra gibberish characters an a Windows

[5]Secure CoPy
[6]Secure SHell File System

text editor. The better text editors on Windows offer options to save files with the appropriate line endings, but it may be easiest to use a terminal-based text editor that runs on the remote machine, such as `nano` or `vim`, if you are forced to work on a Windows machine.

X forwarding

The terminal-based methods described above are limited in that they only allow one to do (and see) what could be done from within the terminal window. GUI-based text editors such as bluefish will not work under this method, and any graphical output of your program will not be displayed.

It is possible to redirect all of the graphical output of the server to your local machine using "X Forwarding". On a Macintosh, this would require that X11 be installed — it is one of the optional installation packages on the system-install DVD. Start X11, which will give you a terminal window on the Macintosh, or open a new terminal window if you're using a Linux machine. Give the command `ssh -X richard.csuchico.edu`, and from then on any graphical windows that would normally open on the Linux machine are instead sent to your local machine. This requires a relatively fast internet connection to be usable.

Under Windows, you will need to use PuTTy and either XWin-32 or XMing. There are good explanations of how to do this on the web: Google "Windows X forwarding" for a helpful set of instructions.

A.6 Where to learn more

The best way to learn Linux quickly is to actually use it, and the best way to make yourself use it is to install it on your own computer. Your instructor will be happy to provide you with a Linux installation disk — just ask! The "Ubuntu" distribution of Linux is well-supported, easy to install, and highly recommended.

To install Ubuntu on your system, merely boot from an Ubuntu disk and follow the installation instructions. The default installation will leave your system in a "dual-boot" configuration, so you can choose at each boot whether to boot into Ubuntu or Windows.

Google is highly useful for learning more about Linux also, and is generally the first place to check if you get stuck on something.

A.7 Problems

A-0 A good password should be a mix of upper-case, lower-case, numeric, and symbol characters. It should be at least 8 characters long, and longer is better. It should *not* be a word in any known language or a name, as such passwords are easily cracked by dictionary attacks.

One excellent method of choosing a password is to use a phrase you can easily remember, then take the first letter of each word in the phrase. For example, "Four score and seven years ago our forefathers..." would become "Fsasyaof". This could be made stronger by replacing letters that looked like numbers with the numbers: "F5a5ya0f". "$5a5ya0f" would be even better — replace the "Four" with a "Capitalized" 4.

Log in and change the password on your account in the lab. Don't use "$5a5ya0f": it's no longer a strong password.

A-1 Create a directory called `Chapter.1` in your home directory in the lab. Create a text file called `numbers.txt` in this directory which contains the numbers 1–9, spelled out, one number per line.

Now use a single unix command to create a file called `bemnrsu.txt` which contains those numbers, alphabetized.

What's that command? Add that command to the end of the `bemnrsu.txt` file, then create a folder named with your last name in the directory /export/classes/phys312/dropbox/ and move the file to that directory.

A-2 Use the `find` command to list all files in your home directory that are less than one day old. (The `man` command will probably be useful here!) What's the command to do this?

A-3 Create the following text file:

```
#!/usr/bin/env python

name = raw_input("What is your name? ")
print("Hello, " + name)
```

Save this file as `hello.py` in your `Chapter.1/` directory. Change the permissions on this file so that it is executable, and execute it. What does it do?

A-4 Experiment with options on the `ls` command. What do the "t", "i", "R", and "F" options do?

A-5 Create a file called `junk` in your home directory. Put some random text in it, then look carefully at the file's listing using `ls -lh`.

Run this command:

```
chmod 200 junk
```

Use `ls -lh` again, and notice the difference. Try viewing the file with `less junk` or `cat junk`. What happens?

Append more lines to the file using `cat >> junk`. (Use ctrl-d to indicate that you're done adding lines.) Does this work? (And how would you tell?)

Run this command:

```
chmod 400 junk
```

Try viewing the file again. Try adding to the file again. Delete the file. What worked, what didn't work? What were the differences? Why?

Spend whatever time is necessary to figure out what's going on, here. The `chmod` command with three digits afterwards is different than what's described in this text, and it's very useful. What would be the permissions resulting from `chmod 664 <filename>`? What 3-digit number would you use to give a file the permissions `rwxr-x--x`?

A-6 Through an inexplicable bureaucratic mix-up, you have been made leader of the local girl-scout troop. Each girl scout has been busily collecting orders for cookies, and has emailed you a text file containing their orders. These text files are stored on the server in the directory `/export/classes/phys312/examples/scouts/`. You want to combine all these text files into one file, sorted alphabetically by customer name.

Combine the files into one sorted file called `orders.txt` in the directory `~/Chapter.1/` using a one-line command. What is the command?

Appendix B

Visual Python

The "Visual Python" package (VPython for short) is a set of routines that allow easy access to 3-D graphics from within Python. VPython was originated by David Scherer in 2000, while he was a sophomore Computer Science major at Carnegie Mellon University, and since then has been used extensively by the physics education community, most notably Bruce Sherwood and Ruth Chabay. [14, 2] As of this writing, VPython is on version 5 and is available for Linux, Macintosh, and Windows.[1]

VPython takes care of all the display management: the drawing of various objects, shading, perspective, and so on. All that the programmer has to do is define the objects and then update their characteristics as needed. The display is updated automatically as soon as the objects' characteristics change.

Example B.-1.1
Write a simulation showing that circular motion is a combination of two simple harmonic motions.

```
#!/usr/bin/env python

"""
Program showing two simple harmonic motions at 90 degrees
to each other, and the resulting circular motion of their
intersection.
"""

from math import *
```

[1] Up-to-date information, including the latest version, is available at `http://vpython.org/`

```
from visual import *

RodWidth = 0.1
R = 1.0                    # Circle radius
SphereRadius = 0.2
theta_o = 0.0
omega = 1.0
dt = 0.01                  # time step
time = 0.0

# Initial setup of system
VerticalRod = cylinder(
    pos=(R*sin(theta_o), -R, 0),
    color=color.red,
    radius=RodWidth,
    axis=(0,2,0),
    opacity=0.6)
HorizontalRod = cylinder(
    pos=(-R, R*cos(theta_o), 0),
    color=color.red,
    radius=RodWidth,
    axis=(2,0,0),
    opacity=0.6)
Dot = sphere(
    pos=(R*sin(theta_o), R*cos(theta_o), 0),
    color=color.yellow,
    radius=SphereRadius,
    opacity=0.5)
Trace = curve(color=color.yellow)

scene.autoscale=False      # prevents automatic rescaling of
                           # display window

# Keep doing this indefinitely
while True:

    rate(100)              # limits speed to 100 frames/sec

    time = time + dt

    # calculate x and y with respect to time.
    x = R*sin(omega*time)
    y = R*cos(omega*time)

    # update positions of objects.
```

```
VerticalRod.x = x
HorizontalRod.y = y
Dot.pos = (x, y, 0)
# and the trace also
Trace.append(Dot.pos)
```

There's a lot of new stuff in this program listing, which is covered in more detail in the following pages.

B.0 VPython Coordinates

The default orientation of the VPython window is x to the right, y up, and z out of the screen towards the viewer.

B.1 VPython Objects

arrow An arrow is like a cylinder, but it has square cross-section and has an arrowhead at the end. Arrows have position (pos), color, axis, and shaftwidth. The position of the arrow is the position of the *tail* of the arrow, and the axis is the vector from the tail to the head. The default value of shaftwidth is $0.1 \times$ (length).

box A box has color, position (pos), length (x extent), height (y extent) and width (z extent). The position of the box refers to the center of the box. The box can also have an axis, which is the vector describing the box's x axis. (by default, the box x axis matches the window's x axis.) In addition, the box has an "up", which is a vector describing the box's y axis. (by default, the boy y ayis matches the window's y ayis.) Between pos, axis, and up it is possible to put a box in any orientation in space. In addition to specifying length, width, and height individually, it is possible to define the size of the box via size=(length, width, height).

cone The cone has color, pos, axis, and radius. The position refers to the center of the cone's base, and the axis is the vector from the position to the apex of the cone. The radius is that of the cone's base.

curve A curve is actually a set of straight line segments between points. It's most often used to trace out the path of some object, such as a

planet or the tip of a gyroscope. Curves have radius, which is the radius of the circular cross-section of the curve. The append function (curve.append(vector(x,y,z)) adds a new point (and line segment) to the end of the curve, which is particularly useful for tracing position of objects. The last line of example B.-1.1, for example, tacks the current position of the Dot onto the curve.

Technical note: No matter how many points there are in a curve, only 1000 points are shown. The points chosen are spread evenly over the whole curve. This keeps the display from slowing down too much, but it also makes the curve turn into a bunch of obviously straight lines if you make the curve very long.

cylinder Cylinders have position (pos), color, radius, and axis. It helps to think of a cylinder as a vector: the position of the cylinder is the location of the tail of the vector, and the axis describes the length and direction of the cylinder. A short cylinder looks like a disk, of course.

ellipsoid An ellipsoid is described by exactly the same parameters as a box: position (pos), color, length, height, width, and axis. (The "up" parameter does not matter, since the ellipsoid is radially symmetric.) The resulting ellipsoid fits exactly within the box with the same dimensions and axis. In addition to specifying length, width, and height individually, it is possible to define the size of the ellipsoid via size=(length, width, height).

frame A frame allows one to form composite objects that can be moved and rotated as if they were one object. For example:

```
pendulum = frame()
cylinder(frame=pendulum,
    pos=(0,0,0)
    radius=0.1
    color=color.blue
    axis=(0,-1,0))
sphere(frame=pendulum,
    pos=(0,-1,0)
    radius=0.2
    color=color.cyan)
pendulum.axis=(0,1,0)    # change orientation of entire frame
pendulum.pos=(1,2,1)     # change position of entire frame
```

You can change the parameters of all objects in a frame with a **for** loop:

```
for item in pendulum.objects:
    item.color=color.green
```

Frame parameters include pos, x, y, z, axis, and up.

helix A helix is a spiral, defined in the same way as a cylinder. It has position (pos), radius, axis, and color, all of which are defined exactly as for a cylinder. In addition, a helix has "coils", which is the number of turns in its entire length, and "thickness" which is the radius of the "wire" that makes the helix.

points A "points" is a set of points which will be indicated by circular or square markers. This is the same as a curve, but without the straight lines between points. The shape of the points can be circular (shape="round") or square (shape="square"). Points also have a characteristic size, which is most often set in pixels.

pyramid A pyramid has a rectangular base and tapers to a point. The pos of the pyramid is the position of the center of the base, and the axis is the vector between base and apex. The height and width of the pyramid are the y and z extent of the base, respectively, and the length can be used to define the x extent of the pyramid. In addition to specifying length, width, and height individually, it is possible to define the size of the pyramid via size=(length, width, height).

ring A ring (toroid) has color, pos, radius, thickness, and axis. The pos of the ring is the position of the center of the ring. The radius is the radius from the center of the ring to the center of the material — the spoke length, if it were a bicycle wheel. Thickness refers to the radius (not diameter) of the ring material. The axis points along the axis of the ring. The magnitude of the axis is irrelevant: only the direction is used to orient the ring in space.

sphere The sphere has position (pos), color, and radius.

vector Vectors don't display, but they're very useful for calculations. They allow correct math, for example: vectors can be added and subtracted with predictable mathematical results. The functions cross(A, B) and dot(A, B) return the cross and dot products of vectors A and B. The mag(A) function returns the magnitude of A. the norm(A) function returns the vector with the same direction as A but with length 1. You can also rotate a vector around an arbitrary axis, using the rotate function described below.

B.2 VPython Controls and Parameters

Here are some of the more important VPython controls and parameters.

color There are nine built-in colors in VPython: red, green, blue, yellow, orange, cyan, magenta, black, and white. These are specified by, for example, "color=color.red". You can also specify RGB values for any color you want:

```
color=color(R,G,B)
```

where R, G, and B are numbers from 0–1 specifying the amount of red, green, and blue in the color.

deleting objects To delete an object, make it invisible. For example, if you have an object called "ball",

```
ball.visible=False
```

display By default, VPython creates a display window called "scene" in which to display your various objects. (This is the "scene" in "scene.autoscale" or "scene.stereo" above.) It is possible to create more than one VPython window with the display function.

```
scene2 = display(title="Second Window")
```

Once this has been done, it's important to specify which display each new object belongs to:

```
ball = sphere(display=scene2, pos= . . . )
```

materials It's possible (with an appropriate video card) to specify the apparent material for an object. Currently-available materials include wood, rough, marble, plastic, earth, diffuse, emmisive (glowing) and unshaded (unaffected by lighting).

```
Home = sphere(material=material.earth)
```

opacity Objects are usually opaque, but the opacity of most objects can be specified with a number from 0.0 (completely transparent) to 1.0 (completely opaque).

```
CrystalBall = sphere(opacity=0.2)
```

rate When you're working on a fast computer, the system can update the picture much faster than you can watch it. The rate(N) function limits the speed of the loop containing it to N frames/second. (See example B.-1.1.)

rotate It's possible to rotate objects around an arbitrary axis using the rotate() function. The following command would rotate "object" counterclockwise through an angle of 45° around a vertical axis located at (1,0,0):

```
object.rotate(angle=pi/4.0, axis=(0,1,0), origin=(1,0,0))
```

The angle must be in radians, and the axis of rotation is defined by the line between origin and origin+axis. If no object axis is specified, rotations are around the object's own pos and axis.

scene The autoscale parameter sets vpython to rescale the window so as to show the entire scene at any time. Sometime this is an annoyance, in which case the line scene.autoscale=False forces VPython to leave the scaling alone.

The stereo parameter describes how 3-D information is displayed. By default, the display is the 2-D projection of the 3-D information on the window. If you want to use the red/blue 3-D glasses, the command scene.stereo="redcyan" changes the scene so that it appears 3-D, at the loss of some color resolution.

B.3 Problems

B-0 Create a ball and a floor, and write a simulation showing the ball bouncing on the floor.

B-1 Write a VPython simulation showing a wheel rolling down a ramp. Make sure the wheel has some sort of spoke or other indicator so that we can tell the wheel is rolling rather than sliding.

B-2 write a VPython simulation that shows the behavior of a damped driven pendulum, as described in equation 5.3. Test it with damping $\beta = 0.5$, $\frac{g}{L} = 1.0$, driving amplitude $A = 1.2$, and driving frequency $\omega = 2/3$.

Appendix C

Least-Squares Fitting

It is very common in physics —and scientific fields in general— to have a set of (x, y) data points for which we want to know the equation $y(x)$. We usually know the *form* of $y(x)$ (linear, quadratic, exponential, power, and so on) but we'd like to know the exact parameters in the function.

The simplest non-trivial case is the linear case, and it turns out that if you solve the linear case you can often solve others with some appropriate algebraic work, so we'll start with the linear case and work from there.

Imagine that you have some data set such as the one shown in figure C.0. Although there's noise in the data, it's clear that the data is linear: $y = ax + b$. The best line through this data would be the line for which the sum of the distances between the line and the data points is a minimum.

As it turns out, though, the distance between the line and the points could be either positive or negative. This means that there are some astonishingly bad lines that would have the same sum of deviations as the "right" line. Taking the sum of the absolute values of the deviations would seem to be better, but the absolute value function causes severe mathematical difficulties when you attempt to take derivatives. The solution to these problems is to minimize the sum of the *squares* of the deviations of the data points from the line.[1]

[1] Thus the name "Least-squares fitting", as you can probably guess.

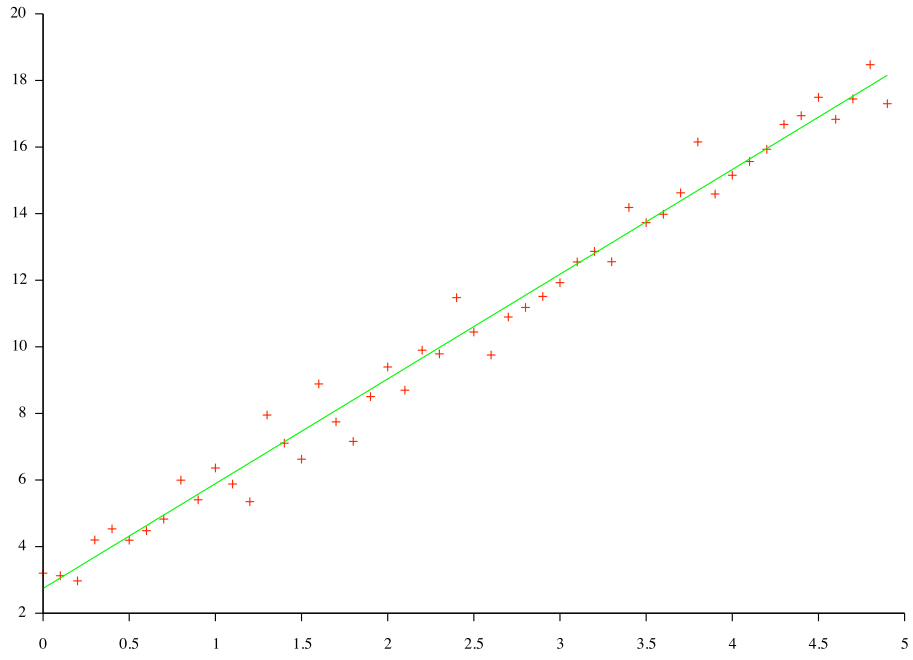


Figure C.0: Somewhat noisy (x, y) data, and the ideal line through that data.

C.0 Derivation

Now let's start some math. We want to minimize the sum of the deviations of the data points from the line. The line is given by

$$y = ax + b \tag{C.1}$$

so the vertical deviation[2] of the i^{th} data point (x_i, y_i) from the line is

$$\delta_i \equiv y_i - (ax_i + b) \; . \tag{C.2}$$

The sum of the squares of all these deviations is then

$$\Delta \equiv \sum_{i=1}^{n} \delta_i^2 = \sum_{i=1}^{n} \left[y_i^2 - 2ax_iy_i - 2by_i + a^2x_i^2 + 2abx_i + b^2 \right] \; . \tag{C.3}$$

[2]Note of course that this *vertical* deviation of a data point from the line is not generally the same as the *perpendicular* distance between the point and the line. An equivalent derivation using the perpendicular distance is considerably more complicated, and gives no significant improvement in results.

We want to minimize Δ, so we take the derivatives of Δ with respect to both a and b and set them equal to zero.

$$\frac{\partial \Delta}{\partial a} = 0 = -2\sum_{i=1}^{n} x_i y_i + 2a\sum_{i=1}^{n} x_i^2 + 2b\sum_{i=1}^{n} x_i \tag{C.4}$$

$$\frac{\partial \Delta}{\partial b} = 0 = -2\sum_{i=1}^{n} y_i + 2a\sum_{i=1}^{n} x_i + 2\sum_{i=1}^{n} b\ . \tag{C.5}$$

At this point, it's helpful to divide both equations by n and make the following substitutions:

$$\overline{x} \equiv \frac{1}{n}\sum_{i=1}^{n} x_i$$
$$\overline{y} \equiv \frac{1}{n}\sum_{i=1}^{n} y_i$$
$$\overline{x^2} \equiv \frac{1}{n}\sum_{i=1}^{n} x_i^2$$
$$\overline{xy} \equiv \frac{1}{n}\sum_{i=1}^{n} x_i y_i$$

Then equations C.4 and C.5 become

$$a\overline{x^2} + b\overline{x} = \overline{xy} \tag{C.6}$$

and

$$a\overline{x} + b = \overline{y}\ . \tag{C.7}$$

Once the averages $\overline{x}$, $\overline{y}$, $\overline{xy}$, and $\overline{x^2}$ are calculated, equations C.6 and C.7 form a system of two equations with two unknowns, a and b. These equations can be written in matrix form as

$$\begin{bmatrix} \overline{x^2} & \overline{x} \\ \overline{x} & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \overline{xy} \\ \overline{y} \end{bmatrix}\ . \tag{C.8}$$

The solution of this matrix equation is

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \overline{x^2} & \overline{x} \\ \overline{x} & 1 \end{bmatrix}^{-1} \begin{bmatrix} \overline{xy} \\ \overline{y} \end{bmatrix} \tag{C.9}$$

where[3]

$$\begin{bmatrix} \overline{x^2} & \overline{x} \\ \overline{x} & 1 \end{bmatrix}^{-1} = \frac{1}{\overline{x^2} - \overline{x}^2} \begin{bmatrix} 1 & -\overline{x} \\ -\overline{x} & \overline{x^2} \end{bmatrix} ,$$

so

$$a = \frac{\overline{xy} - \overline{x}\,\overline{y}}{\overline{x^2} - \overline{x}^2} \tag{C.10}$$

and

$$b = \frac{\overline{x^2}\,\overline{y} - \overline{x}\,\overline{xy}}{\overline{x^2} - \overline{x}^2} . \tag{C.11}$$

We can also calculate our uncertainty in a and b. The deviation of any point (x_i, y_i) from the line $y(x_i)$ is given by

$$\delta_i = y_i - (ax_i + b) \tag{C.12}$$

so the uncertainty in y is [16]:

$$\sigma_y = \sqrt{\frac{\sum \delta_i^2}{n-2}} . \tag{C.13}$$

This leads us to the uncertainty in each parameter (see [1] or equivalent):

$$\sigma_a = \sqrt{\frac{1}{(n-2)} \frac{\overline{\delta^2}}{\overline{x^2} - \overline{x}^2}} \tag{C.14}$$

$$\sigma_b = \sqrt{\frac{1}{(n-2)} \frac{\overline{\delta^2}\,\overline{x^2}}{\overline{x^2} - \overline{x}^2}} \tag{C.15}$$

where $\overline{\delta^2} \equiv \frac{\sum \delta_i^2}{n}$. These values (σ_a, σ_b) are the uncertainties in our slope and intercept, respectively.

As you can imagine, this technique lends itself well to computation. Your program to calculate a least-squares fit merely calculates the various average values $\{\overline{x}, \overline{x^2}, \overline{\delta^2}, \ldots\}$ and then combines these average parameters to report a, b, σ_a, and σ_b as given by equations C.10, C.11, C.14, and C.15.

---

[3]The general equation for the inverse of a 2×2 matrix $M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ is

$$M^{-1} = \frac{1}{|M|} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

C.1 Non-linear fitting

It is possible to extend the least-squares method to non-linear equations. The easiest way of doing this is to linearize the problem, then use linear least-squares fitting.

Example C.1.1
You have data for which you believe the equation to be $y = Ax^B$, and you want to find the values of A and B that best fit this data.

Take the natural log of both sides of the equation:

$$\ln y = \ln A + B \ln x$$

This is a linear equation in the form

$$y' = b + ax'$$

where $y' \equiv \ln y$ and $x' \equiv \ln x$. The intercept b of the least-squares fit line will be $\ln A$, and the slope a corresponds to the power B.

So instead of giving your least-squares function values of x and y, give it values of $\ln x$ and $\ln y$. Once you obtain a and b from the least-squares routine, use

$$A = e^b \qquad B = a$$

to obtain your desired values.

There are other methods of curve fitting for less-convenient equations, in addition to iterative methods which can fit data to equations of any arbitrary form. For more details regarding this area of computation, see [13].

C.2 Python curve-fitting libraries

The scipy library includes a very powerful curve-fitting library, as you might guess. Scipy.optimize.curve_Fit() uses non-linear least-squares fitting to fit a function f to a data set. The method used takes a different approach than the linear least-squares method described above: it takes an initial guess of the parameters and then adjusts them in a reasonably intelligent manner to minimize the squares of the errors.

Here's how to use it.

- Begin by defining the function $f(x)$ that will be fit to the data. The function should take an array of x values and any adjustable parameters as inputs, and should return an array of y values. For example, a power fit function of the form

$$f(x) = ax^b \tag{C.16}$$

would be defined like this:

```
def power(x, a, b):
    f = a*pow(x,b)
    return f
```

- Define an initial guess of the parameters. In the power-fit example from step 1, this would be a list or tuple (or array) containing a and b.

```
guess = (42, 2.1)
```

- Make sure you have your x and y data in array form. (Usually these data arrays are read from a data file.) If you have uncertainties δy in the data points, have those in another array.

- Call curve_fit() with the function name, the x data, the y data, the initial guess, and (optionally) the uncertainties. The routine will return two items: the first is an array containing optimal values of the parameters, the second is a matrix containing the estimated covariance of those optimal values.

```
popt, pcov = curve_fit(power, xdata, ydata, guess, dy)
```

- The elements in popt will be the fitted parameters (a and b in this case), and the square roots of the diagonal elements in pcov will be the uncertainties in each of those elements.

This method is not as fast or as convenient to use as the classic least-squares method, but it can be used to fit equations of almost any form.

C.3 Problems

C-0 Write a least-squares fitting function. The function should accept two lists (or equivalents) containing x and y data points, and should return a tuple containing the slope, the uncertainty in the slope, the intercept, and the uncertainty in the intercept for that data.

C-1 Write a program that reads a tab-delimited text file containing x, y data, sends that data to the function you wrote for problem 1, and prints out the results, neatly formatted. For your convenience, the file `/export/312/examples/LSQ-graph.txt` contains some sample x, y data.

C-2 Use scipy.optimize.curve_fit() to do a linear fit of the same data from problem 1. Compare your results, and the uncertainties in your results.

References

[1] Philip R. Bevington and D. Keith Robinson. *Data Reduction and Error Analysis for the Physical Sciences*. McGraw-Hill, 1992.

[2] Ruth Chabay and Bruce Sherwood. *Matter & Interactions: Electric and Magnetic Interactions*, volume 2. John Wiley and Sons Publishing, 2002.

[3] A. Cromer. Stable solutions using the euler approximation. *American Journal of Physics*, 49:455, 1981.

[4] M. DiStasio and W. C. McHarris. Electrostatic problems? relax! *American Journal of Physics*, 47(5):440–444, May 1979.

[5] David J. Griffiths. *Introduction to Electrodynamics*. Prentice-Hall, Upper Saddle River, third edition, 1999.

[6] Mark Lutz and David Ascher. *Learning Python*. O'Reilly & Associates, 2^{nd} edition, 2004.

[7] Jerry B. Marion and Stephen T. Thornton. *Classical Dynamics*. Harcourt, Brace, Jovanovich, 3^{rd} edition, 1988.

[8] Cameron Newham and Bill Rosenblatt. *Learning the BASH Shell*. O'Reilly & Associates, 2^{nd} edition, 1998.

[9] Mark Newman. *Computational Physics*. 2012.

[10] Tiara Norris, Brendan Diamond, and Eric Ayars. Magnetically coupled rotors. *American Journal of Physics*, 74(9):806–809, September 2006.

[11] Tao Pang. *An Introduction to Computational Physics*. Cambridge University Press, 2nd edition, 2006.

[12] David Park. *Introduction to the Quantum Theory.* McGraw-Hill, third edition, 1992.

[13] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, second edition, 1992.

[14] D. Scherer, P. Dubois, and B. Sherwood. Vpython: 3d interactive scientific graphics for students. *Computing in Science and Engineering*, pages 82–88, Sept/Oct 2000.

[15] Randal L. Schwartz and Tom Phoenix. *Learning Perl.* O'Reilly and Associates, 3^{rd} edition, 2001.

[16] John R. Taylor. *An Introduction to Error Analysis.* University Science Books, 2 edition, 1997.

[17] Shai Vaingast. *Beginning Python Visualization: Crafting Visual Transformation Scripts.* Apress, 2009.